Stratégie de Placement et d'Ordonnancement de Tâches Logicielles pour les Architectures Reconfigurables sous Contrainte Énergétique

Aymen Gammoudi

N° d'ordre : ANNÉE 2018

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

COTUTELLE / ÉCOLE POLYTECHNIQUE DE TUNISIE

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention : Informatique
École doctorale MathSTIC

et

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE DE TUNISIE
Mention : Électronique et Technologie de l'Information et de la Communication
École doctorale EDSA

présentée par

Aymen GAMMOUDI

préparée à l'unité de recherche UMR6074 IRISA-CAIRN
Institut National des Sciences Appliqués et de Technologie-LISI

Stratégie de Placement et d'Ordonnancement de Tâches Logicielles pour les Architectures Reconfigurables sous Contrainte Énergétique

Thèse soutenue à Lannion (Rennes) le 26 juin 2018

devant le jury composé de :

Frank SINGHOFF
Professeur à l'UBO / Président
Anis KOUBAA
Professeur à Prince Sultan University / Rapporteur
Claire PAGETTI
Maître de Recherche à l'ONERA / Rapportrice
Alix MUNIER
Professeur à Sorbonne Université / Examinatrice
Narjes Ben Rajeb ROBBANA
Professeur à l'Université de Carthage / Examinatrice
Daniel CHILLET
Professeur à l'Université de Rennes 1 / Directeur
Mohamed KHALGUI
Maître de Conférence HDR à l'Université de Carthage / Co-directeur
Adel BENZINA
Maître-assistant à l'Université de Carthage / Co-encadrant

"Sois heureux un instant. Cet instant c'est ta vie."

— Omar Khayyâm

Resumé

La conception de systèmes temps-réel embarqués se développe de plus en plus avec l'intégration croissante de fonctionnalités critiques pour les applications de surveillance, notamment dans le domaine biomédical, environnemental, domotique, etc. Le développement de ces systèmes doit relever divers défis en termes de minimisation de la consommation énergétique. Gérer de tels dispositifs embarqués, entièrement autonomes, nécessite cependant de résoudre différents problèmes liés à la quantité d'énergie disponible dans la batterie, à l'ordonnancement temps-réel des tâches qui doivent être exécutées avant leurs échéances, aux scénarios de reconfiguration, particulièrement dans le cas d'ajout de tâches, et à la contrainte de communication pour pouvoir assurer l'échange des messages entre les processeurs, de façon à assurer une autonomie durable jusqu'à la prochaine recharge et ce, tout en maintenant un niveau de qualité de service acceptable du système de traitement.

Pour traiter cette problématique, nous proposons dans ces travaux une stratégie de placement et d'ordonnancement de tâches permettant d'exécuter des applications temps-réel sur une architecture contenant des cœurs hétérogènes. Dans cette thèse, nous avons choisi d'aborder cette problématique de façon incrémentale pour traiter progressivement les problèmes liés aux contraintes temps-réel, énergétique et de communications. Tout d'abord, nous nous intéressons particulièrement à l'ordonnancement des tâches sur une architecture mono-cœur. Nous proposons une stratégie d'ordonnancement basée sur le regroupement des tâches dans des packs pour pouvoir calculer facilement les nouveaux paramètres des tâches afin de réobtenir la faisabilité du système. Puis, nous l'avons étendu pour traiter le cas de l'ordonnancement sur une architecture multi-cœurs homogènes. Finalement, une extension de ce dernier sera réalisée afin d'arriver à l'objectif principal qui est l'ordonnancement des tâches pour les architectures hétérogènes. L'idée est de prendre progressivement en compte des contraintes d'exécution de plus en plus complexes.

Nous formalisons tous les problèmes en utilisant la formulation ILP afin de pouvoir produire des résultats optimaux. L'idée est de pouvoir situer nos solutions proposées par rapport aux solutions optimales produites par un solveur et par rapport aux autres algorithmes de l'état de l'art. Par ailleurs, la validation par simulation des stratégies proposées montre qu'elles engendrent un gain appréciable vis-à-vis des critères considérés importants dans les systèmes embarqués, notamment le coût de la communication entre cœurs et le taux de rejet des tâches.

Mots-clés: Architecture multi-cœur, Reconfiguration, Placement de tâches, Ordonnancement temps-réel, Contrainte énergétique.

Abstract

The design of embedded real-time systems is developing more and more with the increasing integration of critical functionalities for monitoring applications, particularly in the biomedical, environmental, home automation, etc. The developement of these systems faces various challenges particularly in terms of minimizing energy consumption. Managing such autonomous embedded devices, requires solving various problems related to the amount of energy available in the battery and the real-time scheduling of tasks that must be executed before their deadlines, to the reconfiguration scenarios, especially in the case of adding tasks, and to the communication constraint to be able to ensure messages exchange between cores, so as to ensure a lasting autonomy until the next recharge, while maintaining an acceptable level of quality of services for the processing system.

To address this problem, we propose in this work a new strategy of placement and scheduling of tasks to execute real-time applications on an architecture containing heterogeneous cores. In this thesis, we have chosen to tackle this problem in an incremental manner in order to deal progressively with problems related to real-time, energy and communication constraints. First of all, we are particularly interested in the scheduling of tasks for single-core architecture. We propose a new scheduling strategy based on grouping tasks in packs to calculate the new task parameters in order to re-obtain the system feasibility. Then we have extended it to address the scheduling tasks on an homogeneous multi-core architecture. Finally, an extension of the latter will be achieved in order to realize the main objective, which is the scheduling of tasks for the heterogeneous architectures. The idea is to gradually take into account the constraints that are more and more complex.

We formalize the proposed strategy as an optimization problem by using integer linear programming (ILP) and we compare the proposed solutions with the optimal results provided by the CPLEX solver. Inaddition, the validation by simulation of the proposed strategies shows that they generate a respectable gain compared with the criteria considered important in embedded systems, in particular the cost of communication between cores and the rate of new tasks rejection.

Index Terms: Multi-core architecture, Reconfiguration, Task mapping, Real-time scheduling, Energy constraint.

Remerciements

Ce travail de recherche s'est déroulé dans le cadre d'une thèse en cotutelle entre l'École Polytechnique de Tunisie (EPT) et l'Université de Rennes 1. Je tiens à remercier ces deux institutions pour m'avoir offert l'opportunité d'effectuer ma thèse dans des conditions très favorable.

Je remercie Dieu, le Tout Puissant, le Miséricordieux, qui m'a donné l'opportunité de mener à bien ce travail.

Ma reconnaissance s'adresse, en premier, à Monsieur Mohamed KHALGUI, mon Directeur de thèse qui m'a accordé sa confiance dès mon premier projet à l'EPT. Il m'a encadré avec persévérance et dynamisme pendant plusieurs années et m'a guidé dans ce parcours avec pertinence et pragmatisme. Cette thèse n'aurait pas pu voir le jour sans ses encouragements et son soutien.

Je suis redevable à Monsieur Daniel CHILLET, mon co-Directeur de thèse. Je le remercie pour son investissement important dans l'encadrement de mon travail. Sa rigueur et sa maîtrise scientifiques m'ont permis d'apprendre énormément. Je le remercie aussi de m'avoir assuré de bonnes conditions de travail.

Je tiens à remercier également Monsieur Adel BENZINA, maître-assistant à l'EPT, à qui je voudrais exprimer toute ma gratitude. Sa disponibilité, ses conseils éclairés et son soutien m'ont été d'une aide inestimable.

J'adresse ma reconnaissance à Monsieur Anis KOUBAA et Madame Claire PAGETTI, en me faisant l'honneur d'accepter la charge de rapporteur, ainsi que pour leurs remarques très pertinentes sur mon rapport de thèse qui ont contribué à son amélioration.

J'adresse, également, mes remerciements à Madame Alix MUNIER, Madame Narjes Ben Rajeb ROBBANA et Monsieur Frank SINGHOFF d'avoir accepté examiner ce travail.

Je remercie également Monsieur Olivier SENTIEYS, Professeur des Universités à l'Université de Rennes 1 et responsable de l'équipe CAIRN, pour m'avoir accueilli dans son équipe de recherche.

Je remercie particulièrement mon oncle Mohamed Mohsen GAMMOUDI, Professeur à l'université de la Manouba, qui m'a appris un jour que c'est avec le courage et la volonté qu'on peut atteindre les objectifs visés.

Je tiens aussi à mentionner le plaisir que j'ai eu à travailler au sein l'équipe CAIRN, et j'en remercie ici tous les membres, permanents et doctorants, qui m'ont accompagné pendant ce travail de thèse.

Je remercie également tous les membres du laboratoire LISI-INSAT pour leur accueil et leur disponibilité, en particulier le directeur Monsieur Moncef GASMI.

Je remercie fortement ma chère mère, mon cher père, mes frères ainsi que toute ma grande famille pour leurs prières et leur soutien. Une pensée particulière à mes correcteurs de fautes d'orthographes de ce manuscrit, mon cousin Aymen GAMMOUDI "Barhoumi" et mes amis Rania AYACHI, Mohamed Oussama BENSALEM et Farid ADAILI.

Enfin merci de tout mon cœur à ma fiancée Sabrine, qui a encaissé ces 3 années avec beaucoup de patience et d'amour, et qui m'a soutenu chaque jour; sans toi je ne sais pas dans quel état j'aurai fini !

Contents

Liste des Acronymes

MPSoC	Multi-Processor System-on-Chip
CPU	Central Processing Unit
DSP	Digital Signal Processor
ASIC	Application-Specific Integrated Circuit
FPGA	Field-Programmable Gate Array
CMOS	Complementary Metal Oxide Semiconductor
EDeg	Earliest Deadline with energy guarantee
WCET	Worst Case Execution Time
MPEG	Moving Picture Experts Group
QoS	Quality of Service
IoT	Internet of Things
RM	Rate Monotonic
DM	Deadline Monotonic
EDF	Earliest Deadline First
LLF	Least-Laxity First
DVS	Dynamic Voltage Scaling
DVFS	Dynamic Voltage Frequency Scaling
NoC	Network-on-Chip
B&B	Branch and Bound
LP	Linear Programming
ILP	Integer Linear Programming
DAG	Directed Acyclic Graph
RTSys	Real-Time System
RTConst	Real-Time Constraints
EnergyConst	Energy Constraint
CommConst	Communication Constraint
UML	Unified Modeling Language
P2P	Point-2-Point
WCTT	Worst Case Transmission Time

CL	Cluster
LCL	List of Cluster
CP	Common Processor
CPCL	Common Processor of Cluster
LCPCL	List of CPCL
DT	Data Traffic
OS	Operating System
RTOS	Real-Time Operating System
HAL	Hardware Abstraction Layer
ISR	Interrupt Service Routine

List of Figures

List of Tables

Chapitre 1

Introduction générale

1.1 Contexte et problématique de la thèse

Les dispositifs embarqués, comme les réseaux de capteurs communiquant sans fil, offrent des possibilités extrêmement intéressantes pour les applications de surveillance, notamment dans le domaine biomédical, environnemental, domotique, etc [3–6]. Ils fonctionnent grâce à des batteries ou/et des supercondensateurs qui se rechargent périodiquement. En effet, les capteurs qui constituent chaque nœud du réseau ne disposent pas de ressources énergétiques permanentes et leur autonomie énergétique sur de longues périodes est un problème [5]. Dans ces conditions, la minimisation de l'énergie est devenue une préoccupation majeure sachant que les nouvelles générations de ces systèmes proposent des capacités de calcul de plus en plus importantes tout en utilisant des batteries à charge limitée.

Parallèlement, on observe qu'à mesure que les systèmes embarqués deviennent de plus en plus complexes, la plate-forme d'exécution repose sur des architectures multiprocesseurs reconfigurables de par le bénéfice qu'elles apportent par la rapidité d'adaptation et la grande capacité de calcul qu'elles offrent [2, 7–11]. Ces systèmes sont généralement amenés à gérer des tâches critiques, mais également des tâches dont la qualité de service peut être dégradée sans que cela ne conduise à des dangers pour le matériel ou pour les humains [2, 7, 9]. Toutefois, ce contexte exige la mise en place de différents modes de fonctionnement et de techniques de fiabilité, d'où la notion de systèmes embarqués temps-réel reconfigurables dynamiquement [7, 12]. Il existe deux types de reconfiguration

dynamique : la reconfiguration manuelle [11] appliquée par l'utilisateur et la reconfiguration automatique [13] qui est appliquée par l'intervention automatique de programmes ou bien par des agents intelligents. Qu'ils soient manuels ou automatiques, ces différents changements dynamiques de comportement placent les systèmes face à des scénarios de reconfiguration de type ajout et/ou suppression de tâches. Dans le cas d'un scénario de reconfiguration qui supprime des tâches, l'utilisation du processeur diminue et le nouvel état du système ne peut, à priori, pas conduire à la violation des contraintes. Cependant, après l'ajout de tâches, il est possible que certaines contraintes temps-réel et/ou énergétiques soient violées. Bien que le cas de la suppression de tâches puisse avoir de l'intérêt, nous avons focalisé nos travaux sur les cas d'ajout de tâches qui sont ceux qui vont conduire le système dans des états non acceptables. Nous verrons toutefois en perspectives que le cas de suppression des tâches pourrait venir compléter notre étude et permettre une amélioration de la qualité de service.

Gérer de tels dispositifs embarqués, entièrement autonomes, nécessite cependant de résoudre différents problèmes liés à la quantité d'énergie disponible dans la batterie, à l'ordonnancement temps-réel des tâches qui doivent être exécutées avant leurs échéances, aux scénarios de reconfiguration surtout dans le cas d'ajout de tâches et à la contrainte de communication pour pouvoir assurer l'échange des messages entre les processeurs, de façon à assurer une autonomie durable jusqu'à la prochaine recharge et ce, tout en maintenant un niveau de qualité de service acceptable du système de traitement.

1.2 Contributions de la thèse

L'objectif principal de cette thèse consiste à proposer une stratégie de placement et d'ordonnancement de tâches permettant d'exécuter des applications temps-réel sur une architecture reconfigurable contenant des cœurs hétérogènes, en vue de gérer la quantité d'énergie disponible dans la batterie et de garantir le fonctionnement du système jusqu'à la prochaine recharge. Elle vise également à assurer que la communication entre les différents cœurs soit toujours faisable et que tous les messages peuvent arriver à leur destination en satisfaisant les contraintes temporelles.

Attaquer cette problématique directement n'est pas une tâche évidente étant donné la complexité des contraintes à respecter. Aussi, nous avons choisi d'aborder la problématique de façon incrémentale pour prendre en compte (progressivement) la complexité des architectures. Tout d'abord, nous nous intéressons à l'ordonnancement des tâches pour l'architecture mono-cœur. Par la suite, nous avons adressé le cas des architectures multi-cœurs homogènes en étendant les propositions faite dans le cas mono-cœur. Finalement, une extension de notre méthode a été développée pour le cas des architectures hétérogènes qui sont majoritaires de nos jours. L'idée est de prendre progressivement en compte des contraintes d'exécution plus en plus complexes. La méthodologie que nous avons suivi peut donc être représentée par la Figure 1.1.

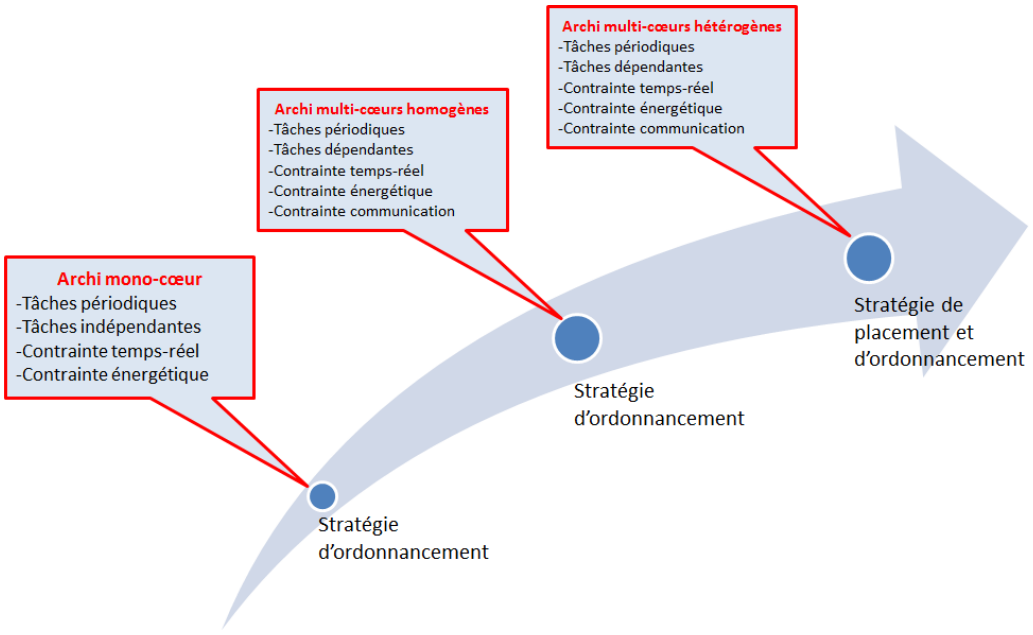

FIGURE 1.1: Progression incrémentale de la résolution du problème de thèse.

Dans l'ensemble de nos travaux, nous nous basons sur le modèle de tâches proposé par Marinoni Mauro et Buttazzo Giorgio [14] qui est caractérisé par l'élasticité des tâches et qui est très utilisé surtout dans les systèmes temps-réel à contraintes souples [2, 7, 9, 15–20]. Une tâche est dite élastique (ou flexible) si elle dispose d'une période pouvant être modifiée dans un intervalle fourni par le concepteur logiciel. Dans ce contexte, pour satisfaire la contrainte temps-réel et/ou celle énergétique, Wang et al. dans [2, 15] s'intéressent à la réadaptation des tâches en modifiant leurs paramètres afin de réduire le taux d'utilisation du processeur. Chaque modification d'un paramètre a un coût qui est

calculé par la différence entre la nouvelle valeur et la valeur initiale du paramètre. Toutefois, la modification des paramètres est souvent lourde. En d'autres termes, le coût de modification dans certains cas est très élevé et, de plus, cela conduit à une dégradation plus importante du système. Pour cela et afin de réduire le coût de modification, nous proposons des stratégies d'ordonnancement qui permettent de prendre en compte une notion de flexibilité exprimée dans le modèle de tâches et permettant d'adapter l'exécution de l'application/tâches dynamiquement en fonction des ressources disponibles. Les principales contributions de cette thèse sont :

- Proposition d'une stratégie d'ordonnancement, basée sur l'élasticité des tâches, pour les architectures mono-cœur. Cette stratégie est valable pour les architectures reconfigurables dynamiquement et elle permet de garantir la faisabilité du système après la violation de l'une des contraintes temps-réel et/ou énergétiques. L'objectif de la stratégie proposée est d'assurer l'ordonnancement des tâches, sous les contraintes précitées, en adaptant leurs temps d'exécution et leurs périodes.

- Réalisation d'un outil de simulation qui permet d'évaluer la stratégie proposée par rapport à l'algorithme proposé par Wang et al. [2, 15]. Les résultats montrent que la stratégie proposée permet de réduire le coût de modification des paramètres par rapport à l'approche dans [2, 15]. En terme de gain et sur un ensemble de systèmes générés aléatoirement, notre stratégie offre 27% de gain de coût de modification par rapport à la même approche précitée.

- Proposition d'une stratégie d'ordonnancement adressant les architectures multi-cœurs homogènes. Dans cette architecture, les tâches sont ordonnancées sous les trois contraintes suivantes : temps-réel, énergétique et communication.

- Proposition d'une extension de la stratégie précédente qui s'adresse aux architectures multi-cœurs hétérogènes. Elle propose de placer et d'ordonnancer les tâches après un événement de configuration ou de reconfiguration tout en garantissant la faisabilité du système. Les résultats expérimentaux sur notre stratégie montrent qu'elle offre un gain du coût de communication de 11% par rapport à la stratégie proposée par Govil et al. [21] et de 7% par rapport à l'algorithme proposé par Bhardwaj et al. [22]. Par ailleurs, nous avons utilisé une autre métrique de comparaison qui est le taux de rejet des tâches et nous avons montré que la stratégie

proposée permet de réduire le taux de rejet de 62% par rapport à l'algorithme [21] et de 59% par rapport à celui présenté dans [22].

1.3 Plan de la thèse

Le manuscrit de thèse suivra l'approche incrémentale que nous a conduit durant nos travaux et il se compose des chapitres suivants :

Chapitre 2 : ce chapitre présente l'architecture générale des systèmes embarqués cibles. Ce chapitre dresse un état de l'art sur les systèmes temps-réel et plus précisément sur l'ordonnancement de tâches élastiques (flexibles) sous contraintes temps-réel et énergétiques. Nous évoquons aussi les différentes stratégies d'ordonnancement pour les architectures mono et multi-cœurs. Quant aux méthodes et les algorithmes d'optimisation pour le placement des tâches, ils seront également détaillés dans ce chapitre.

Chapitre 3 : ce chapitre décrit les modèles de tâches, de consommation énergétique étudiés ainsi que le problème de reconfiguration sous-jacent. Nous y proposons des solutions d'ordonnancement basées sur le regroupement des tâches dans des packs afin de réobtenir la faisabilité du système, qui est composé d'un seul cœur. L'ensemble des solutions proposées constituera la stratégie d'ordonnancement.

Chapitre 4 : ce chapitre traite principalement le cas des architectures multi-cœurs hétérogènes. L'objectif de ce chapitre consiste à proposer une stratégie de placement et d'ordonnancement des tâches en minimisant le coût total de communication. Notre stratégie vise dans un premier temps à placer les tâches qui s'échangent des données sur le même cœur lorsque cela est possible ou de réduire la distance entre ces tâches si le placement sur le même cœur n'est pas possible. Dans un deuxième temps, elle vérifie si les trois contraintes (temps-réel, énergétique et communication) sont respectées. S'il y a au moins une contrainte violée après un événement d'ajout de tâches, la stratégie propose d'appliquer des solutions afin de réobtenir la faisabilité du système. L'ordonnancement de tâches sur une architecture homogène est un cas particulier d'une architecture hétérogène et elle est traitée dans ce chapitre.

Chapitre 5 : ce chapitre conclut la thèse par un récapitulatif des contributions et propose une discussion sur une série de perspectives d'extensions possibles pour montrer l'utilité de nos approches dans nos futurs travaux de recherche.

Chapitre 2

État de l'art

2.1 Introduction

Dans ce chapitre seront présentés les travaux de recherche qui traitent l'ordonnancement temps-réel dans un contexte de systèmes embarqués reconfigurables mono et multi-cœurs. Nous présentons dans la première section l'architecture globale des systèmes embarqués ainsi que la notion de reconfiguration. La deuxième section est consacrée à présenter les différents algorithmes d'ordonnancement temps-réel pour les architectures mono et multi-cœurs, notamment les algorithmes qui gèrent la consommation énergétique. Quant aux méthodes et aux algorithmes d'optimisation pour le placement des tâches, ils seront détaillés dans la dernière partie.

2.2 Architecture matérielle des systèmes embarqués

Un système embarqué est un système intégré dans un système plus large (d'où l'appellation "embedded systems") avec lequel il est interfacé, et pour lequel il réalise des fonctions particulières (contrôle, surveillance, communication, etc) [23, 24]. Nous présentons dans cette section l'architecture matérielle des systèmes embarqués ainsi qu'une étude bibliographique sur la gestion énergétique de ce genre de systèmes [2, 7–11].

2.2.1 MPSoC (*MultiProcessor System-on-Chip*)

Un système multiprocesseur sur puce (ou *multiprocessor system-on-chip* MPSoC) [25, 26] est un système embarqué complet intégrant sur une seule puce de silicium plusieurs composants complexes et hétérogènes. Ces composants peuvent être des unités de calcul spécifiques programmables et/ou non programmables (CPU, DSP, ASIC, FPGA), des réseaux de communication complexes (bus hiérarchiques sur puce, réseau sur puce), des composants de mémorisation variés ou encore des périphériques E/S, etc. La Figure 2.1 représente un modèle générique d'un système MPSoC hétérogène avec des parties matérielles et logicielles structurées en couches. Le matériel se divise en deux couches :

- La couche basse contient les composants de calcul et de mémorisation utilisés par le système (µP, µC, DSP, matériel dédié IPs, mémoires) ; et
- La couche matérielle de communication embarquée sur la puce composée de deux sous-couches : réseau de communication (liens point-à-point, bus hiérarchique, réseau sur puce) et adaptateurs de communication entre le réseau et les composants de la première couche.

Le logiciel embarqué est aussi découpé en couches :

- La couche la plus basse est l'abstraction du matériel HAL (*Hardware Abstraction Layer*) permet de faire le lien avec le matériel en implémentant les pilotes E/S des périphériques, des contrôleurs de composants, les routines d'interruption ISR (*Interrupt Service Routine*) ;
- La couche système d'exploitation qui permet de porter l'application sur l'architecture (gestion de ressources, communication et synchronisation, ordonnancement).
- La couche application.

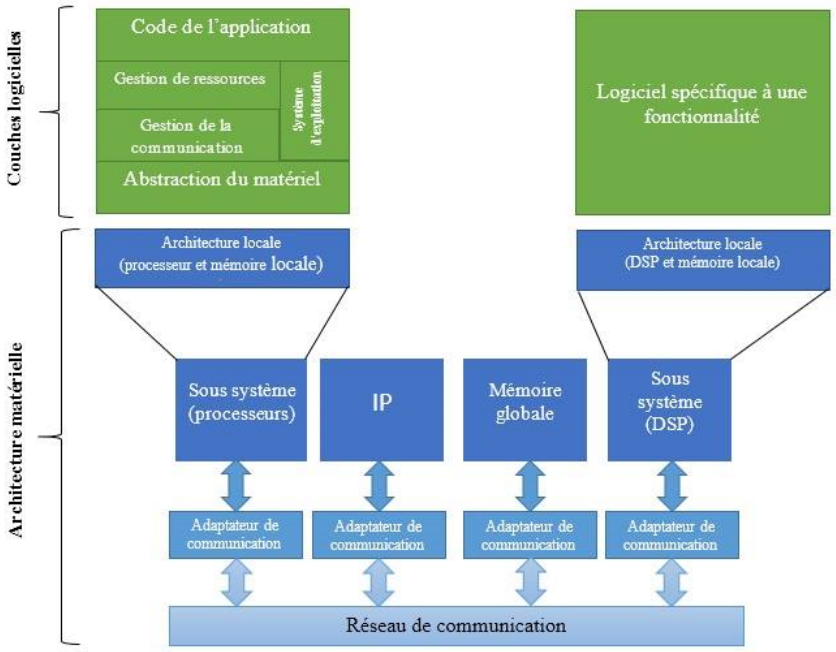

FIGURE 2.1: Architecture d'un système MPSoC hétérogène [1].

Les ASICs et les FPGAs sont des circuits intégrés que l'on peut configurer pour réaliser à bas coût des fonctions "câblées" simples dont les temps de réaction sont extrêmement courts.

2.2.2 Consommation d'énergie pour les systèmes embarqués

Pour la haute efficacité des MPSoC, nous présentons dans cette sous section, une évaluation de la consommation de puissance. Dans la littérature [2, 15, 27, 28], la puissance consommée P dans les circuits possédant la technologie CMOS (*complementary metal oxide semi-conductor*) est souvent divisée en deux composantes : une composante dynamique due à l'activité des transistors et une composante statique due à leurs courants de fuites. Ces courants de fuites sont considérés négligeables dans cette étude. La puissance dynamique est composée elle même de deux parties : la puissance dissipé par le courant de court-circuit et la puissance dissipé par le courant de commutation. Ces deux courants apparaissent lors de la commutation des transistors.

Les auteurs de [2, 15, 28] ont prouvé par une démonstration complète [28] que la puissance P est proportionnelle au carré de l'utilisation du processeur U, i.e., $P = k * U^2$.

Cette forumle est utilisée dans de nombreux travaux traitant la problématique de l'ordonnancement temps-réel sous contrainte énergétique [2, 9, 17, 20, 29–31].

Cette formule est intéressante et sera utilisée dans cette thèse pour contrôler la consommation énergétique en variant l'utilisation du processeur.

2.2.3 Systèmes embarqués rechargeables

La majorité des systèmes embarqués est alimentée par une batterie ou un supercondensateur. Un capteur sans fil est un exemple concret d'un système embarqué qui peut être à la fois rechargeable et temps-réel. Un réseau de capteurs est un système autonome [32] et il est capable de comprendre plusieurs types de capteurs pour mesurer des paramètres de l'environnement tels que la température, l'humidité, la force, le bruit, etc. Les réseaux de capteurs sans fil sont utilisés dans de nombreux domaines :

- Applications environnementales : la détection automatique des incendies et feux de forêts, la détection des tremblements de terre, le suivi des trajectoires des animaux, etc.
- Applications médicales : le médecin peut recevoir les données physiologiques de son patient grâce aux capteurs embarqués sur le corps de ce dernier. Il peut par la suite réaliser le diagnostic nécessaire.
- Applications de domotique (maison intelligente) : Une personne peut contrôler sa maison en installant des capteurs dans toutes les pièces. Elle peut donc superviser la régulation du chauffage, l'éclairage des chambres, etc.

Si les capteurs sont alimentés par une batterie, la durée de vie de celle-ci déterminera la durée de vie du capteur. La notion d'énergie est donc une contrainte forte. L'objectif du concepteur est de fournir la plus longue autonomie au capteur tout en respectant la contrainte liée à la taille physique du système.

La Figure 2.2 montre quelques produits qui sont rechargeables par l'énergie solaire. L'énergie récupérée par les panneaux photovoltaiques sera stockée dans une ou plusieurs batteries [3, 4, 33].

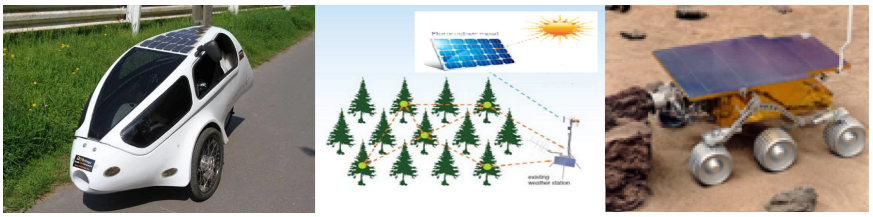

FIGURE 2.2: Produits profitant de l'énergie solaire.

Dans le cas d'un réseau de capteurs sans fils qui surveille les départs de feu dans une forêt [3], les capteurs sont alimentés par une batterie qui est rechargeable périodiquement par l'énergie solaire (Figure 2.3). L'exécution de l'application de surveillance en journée (en présence du soleil) ne viole pas la contrainte énergétique. Cependant, la contrainte peut être violée la nuit à cause de l'épuisement de la batterie et, par conséquent, le système ne pourra plus assurer l'exécution de l'application/tâches.

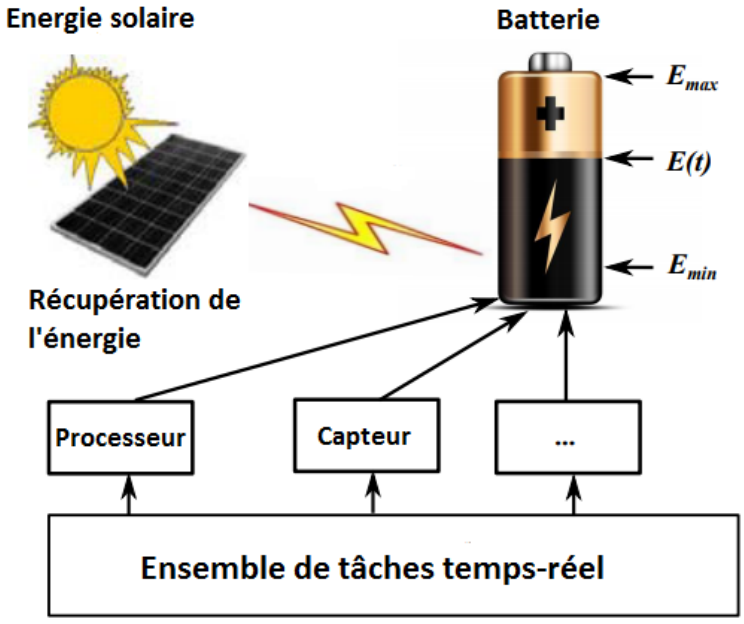

FIGURE 2.3: Modèle de batterie rechargeable récupérant de l'énergie solaire.

Il est donc important de garantir que le système s'exécute jusqu'à la prochaine recharge pour éviter des dégâts importants. Ainsi, de nombreuses recherches scientifiques [2, 5, 9, 34, 35] se sont intéressées à la gestion des cycles de recharges de batteries en utilisant la technique d'ordonnancement des tâches. Chetto et al [5] proposent un algorithme d'ordonnancement nommé EDeg (*Earliest Deadline with energy guarantee*) permettant d'ordonnancer des tâches périodiques sous contrainte énergétique. Les auteurs de [5]

considèrent un système composé d'un seul cœur de calcul qui est alimenté par un réservoir d'énergie. Ce réservoir d'énergie est rechargé par une source d'énergie renouvelable. Le cœur doit être capable d'exécuter une configuration de tâches périodiques temps-réel strictes tout en considérant leur demande énergétique, leur échéance et l'énergie disponible dans le réservoir. Ainsi à chaque instant, la tâche ayant la plus proche échéance parmi les tâches prêtes est exécutée uniquement si d'une part le réservoir n'est pas vide et si d'autre part il y a suffisamment d'énergie dans le système de sorte que l'exécution de celle-ci ne compromette pas l'exécution de tâches périodiques futures. Dans le cas contraire, c'est-à-dire s'il n'y a pas assez d'énergie dans le système ou que le réservoir est vide, le cœur doit être mis en mode veille pendant une durée t_{veille} afin de recharger la batterie. L'algorithme EDeg a fait l'objet d'une étude de performance exhaustive publiée dans [35, 36]. Néanmoins, dans des applications réelles comme, le cas d'une batterie alimentée par une source d'énergie photovoltaïque, pendant la nuit la puissance reçue est nulle et EDeg n'est pas capable de garantir l'exécution du système jusqu'à la prochaine recharge. Par conséquent, il faut pouvoir gérer la quantité d'énergie disponible dans la batterie sans engendrer la panne du système. En effet, nous ne considérons pas dans cette thèse une configuration de tâches périodiques temps réel strictes mais une configuration de tâches périodiques temps-réel souples qui tolèrent un seuil de pertes en-deçà duquel les performances du système sont certes dégradées en adaptant les paramètres de tâches mais le résultat est toujours jugé acceptable.

Dans cette thèse, nous nous intéressons aux techniques de gestion énergétique basées sur l'ordonnancement et l'adaptation des paramètres des tâches.

2.2.4 Gestion de l'énergie pour les systèmes reconfigurables et rechargeables

Les systèmes embarqués rechargeables peuvent être de grande taille, distribués et évoluer dans un environnement dynamique [7, 10, 12]. Ils sont généralement amenés à gérer des tâches critiques, mais également des tâches dont la qualité de service peut être dégradée sans que cela ne conduise à des dangers pour le matériel ou pour les humains. Toutefois, ce contexte exige la mise en place de différents modes de fonctionnement et de techniques de fiabilité, d'où la notion de systèmes embarqués reconfigurables dynamiquement [7, 10, 12]. Il existe deux types de reconfigurations dynamiques : la reconfiguration

manuelle [11] appliquée par l'utilisateur et la reconfiguration automatique [13] qui est appliquée par l'intervention automatique de programmes ou bien par des agents intelligents. Qu'ils soient manuels ou automatiques, ces différents changements dynamiques de comportement placent les systèmes face à des scénarios de reconfiguration de type ajout et/ou suppression de tâches. Dans ces conditions, il est possible qu'à la suite de plusieurs scénarios de reconfigurations, certaines contraintes temps-réel soient violées [7, 16]. De plus, ce type d'événement peut violer également la contrainte énergétique pour les systèmes rechargeables périodiquement, notamment les systèmes embarqués gérant des énergies renouvelables [9, 20]. Dans les travaux [7, 9, 16, 20] et afin de garantir que le système ne viole pas ces contraintes, les auteurs s'intéressent à la réadaptation des tâches en modifiant leurs paramètres après le scénario de reconfiguration. Dans ce contexte, il existe des travaux qui s'intéressent à la modification des *deadlines* des tâches [31, 37, 38] afin de réduire l'utilisation du processeur pour que le système soit de nouveau faisable. D'autres travaux s'articulent autour de la modification des périodes des tâches [2, 7, 15]. Bien que ces travaux peuvent réobtenir la faisabilité du système, d'autres travaux proposent différentes approches pour changer la vitesse du processeur en diminuant les WCETs des tâches [2, 16, 39]. Chaque modification a un coût $CoûtModif_{\tau_i}$ qui est calculé par la différence entre la nouvelle valeur et la valeur initiale du paramètre. Toutes les solutions proposées permettent de satisfaire les contraintes violées après un ou plusieurs scénarios de reconfiguration. Toutefois, la modification des paramètres est souvent lourde. En d'autres termes, le coût de modification dans certains cas est très élevé et, de plus, cela conduit à une dégradation plus importante du système. Nous adressons donc cette problématique dans le chapitre suivant.

Nous nous intéressons dans cette thèse aux systèmes reconfigurables dynamiquement et nous définissons une reconfiguration comme toute opération permettant l'ajout de tâches durant l'exécution de l'application.

2.3 Systèmes temps-réel

Beaucoup de systèmes embarqués sont de type temps-réel, c'est-à-dire qu'ils exécutent des tâches sous contraintes temporelles. Dans la littérature [1, 3, 9, 29, 40], de nombreuses définitions ont été proposées pour la notion de systèmes temps-réel. La définition la plus utilisée est celle de Stankovic et al. [41] qui définit les systèmes temps-réel comme *des*

systèmes informatiques qui doivent obligatoirement réagir à l'environnement dans les contraintes précises du temps. En conséquence, le comportement correct de ces systèmes dépend non seulement de la valeur du résultat logique, mais aussi du temps de réponse auquel le résultat est produit. En d'autres termes, un système temps-réel doit satisfaire non seulement aux contraintes logiques, mais également aux contraintes temporelles qui lui sont imposées. Réagir trop tard peut conduire à des conséquences catastrophiques pour le système lui-même ou le processus. En effet, le respect des contraintes temporelles est la principale contrainte à satisfaire.

2.3.1 Taxonomie des systèmes temps-réel

Nous pouvons classer les systèmes temps-réel selon le niveau de criticité de leurs contraintes temporelles. Nous distinguons alors les systèmes :

Temps-réel dur (en anglais *hard*) : c'est un système soumis à des contraintes temporelles strictes, c'est-à-dire pour lequel la moindre faute temporelle peut avoir des conséquences humaines ou matérielles catastrophiques. Plusieurs applications dans les domaines avioniques, automobile, etc., sont temps-réel dur ;

Temps-réel souple (en anglais *soft*) : c'est un système pour lequel un certain nombre de fautes temporelles peut être toléré. On parle alors de qualité de service et de dégradation du système ;

Temps-réel ferme (en anglais *firm*) : C'est une sous-classe du temps-réel souple pour laquelle le manquement occasionnel des échéances est autorisé. Ce type de système tolère donc le dépassement des échéances mais à la différence des systèmes à contraintes soft, ces dépassements sont quantifiés.

2.3.2 Tâches temps-réel

Du point de vue du processeur, une tâche est une activité qui consomme des ressources de la machine informatique, notamment de la mémoire et du temps CPU. Une application temps-réel est composée d'un ensemble de tâches [4, 27, 42]. Une tâche correspond à l'exécution d'une séquence d'opérations données durant la vie du système. Nous pouvons citer comme exemple une tâche qui relève régulièrement la température d'un capteur.

Nous appelons alors *job* d'une tâche, une occurrence d'exécution de celle-ci. Ainsi, une tâche est constituée d'un ensemble de *jobs*.

Un système temps-réel est constitué d'un ensemble de tâches temps-réel soumises à des contraintes temps-réel. Une tâche temps-réel peut être :

- **Périodique :** ses *jobs* se répètent indéfiniment et il existe une durée constante entre deux activations de *jobs* successives appelée période,
- **Sporadique :** ses *job* se répètent indéfiniment et il existe une durée minimale entre deux activations de *jobs* successives,
- **Apériodique :** il n'y a pas de corrélation entre deux *jobs* successifs d'une même tâche.

2.3.2.1 Etat et gestion des tâches

Dans un environnement multitâche, à tout instant, chaque tâche peut être dans l'un des états suivants :

- **Élue :** si la tâche est en train de s'exécuter.
- **Prête :** si la tâche est en attente de pouvoir s'exécuter. La priorité des tâches prêtes est inférieure ou égale à la priorité de la tâche élue.
- **Suspendue :** c'est le cas des tâches qui sont en attente d'un événement qui provoquera leur réveil.

La Figure 2.4 présente les différents états d'une tâche ainsi que les transitions d'un état à l'autre.

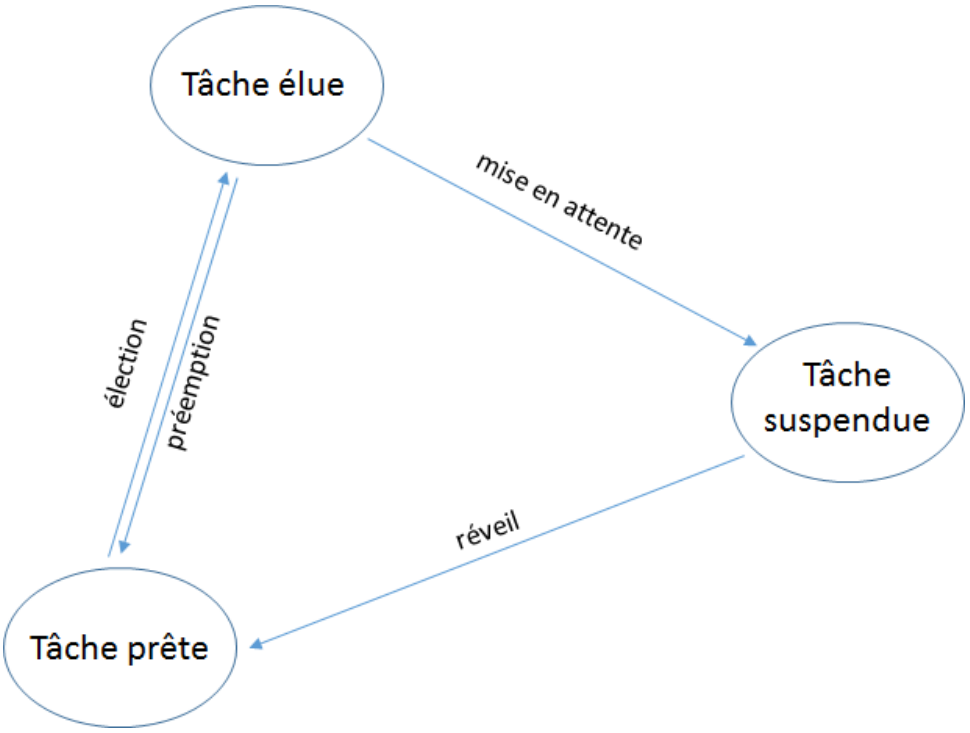

FIGURE 2.4: différents états d'une tâche.

2.3.2.2 Tâches élastiques (flexibles)

Dans la littérature, Mauro Marinoni et Giorgio Buttazzo [14, 43–45], proposent un nouveau modèle de tâches basé sur les tâches élastiques (ou flexibles). Une tâche est dite élastique si elle dispose d'une période pouvant être modifiée dans un intervalle fourni par le concepteur logiciel. Dans ce contexte, chaque tâche est considérée comme flexible, donc l'utilisation du processeur peut être modifiée en changeant les périodes de tâches dans les plages spécifiées. Plus précisément, chaque tâche τ_i est caractérisée par quatre paramètres : i) un WCET W_i, qui dépend de la vitesse du processeur, ii) une période minimale $T_{i_{min}}$ (considérée comme la période nominale T_i) et iii) une période maximale $T_{i_{max}}$ que τ_i peut accepter. Ce dernier permet de spécifier la flexibilité de la tâche pour varier son taux d'utilisation (section 3.2.2) et ainsi adapter le système à une nouvelle configuration.

Ce modèle de tâches a été utilisé dans divers domaines où le respect des contraintes temporelles est souhaitable, mais le dépassement d'une contrainte dans certaines limites est toléré et ne rend pas le système inexploitable, notamment dans les domaines suivants :

- systèmes multimédia [43] : Les tâches qui traitent l'échantillonnage de la voix, l'acquisition d'images, la génération de sons, la compression de données et la lecture de vidéos sont exécutées périodiquement, mais leurs taux d'exécution ne sont pas aussi rigides que dans les applications de contrôle. L'échéance d'une *deadline* lors de l'affichage d'une vidéo MPEG peut diminuer la qualité de service (QoS), mais n'entraîne pas de problème catastrophique. En fonction de la QoS demandée, le taux d'utilisation de tâches peut être augmenté ou diminué par l'OS pour s'adapter aux exigences d'autres activités concurrentes.

- Internet des objets (*Internet of things* IoT) [46] : En considérant les applications de domotique, certaines tâches permettant de contrôler le chauffage, l'éclairage des pièces, etc, peuvent être retardées afin d'exécuter d'autres dont la contrainte temporelle est dure, notamment la tâche d'alarme en cas d'incendie.

- systèmes reconfigurables : Le nombre de tâches qui s'exécutent sur ce système varie d'une reconfiguration à une autre. Les auteurs dans [7, 15–19] utilisent le modèle de tâches élastiques pour faire varier l'utilisation du processeur (en faisant varier le taux d'utilisation des tâches) afin de respecter les contraintes temps-réel et/ou énergétiques. Dans d'autres situations, la possibilité d'adapter le taux d'utilisation du processeur, en allongeant les périodes de tâches, augmente la flexibilité du système dans la gestion des conditions de surcharge. Par exemple, lorsqu'une nouvelle tâche ne peut pas être exécutée par le système, au lieu de la rejeter, le système peut essayer de réduire le taux d'utilisation du processeur par les autres tâches (en augmentant leurs périodes de façon contrôlée) afin de réduire l'utilisation totale du processeur.

Ce modèle possède l'avantage de résoudre la violation des contraintes après un scénario de reconfiguration. Nous utiliserons donc ce modèle pour nos travaux d'ordonnancement et de placement de tâches.

2.3.2.3 Migration de tâches

Pour les systèmes multiprocesseurs temps-réel, la notion de migration de tâches est souvent non prise en compte. Ceci est dû au fait que d'une part, déplacer un *job* assigné à un processeur donné vers un autre processeur est coûteux en temps. D'autre part,

la théorie de l'ordonnancement temps-réel multiprocesseurs ainsi que les outils utilisés, ont été jusqu'à présent développés pour des approches par partitionnement où la migration n'est pas tolérée. En plus de leur classification par priorités, Carpenter et al. [47] classent également les ordonnancements temps-réel multiprocesseurs selon les possibilités de migration :

Sans migration : Cette classe n'autorise aucune migration. Est ici classée l'approche par partitionnement (voir Section 2.3.4.2).

Migration restreinte : La migration dans cette classe n'est autorisée qu'au niveau *jobs*. Celui-ci doit donc s'exécuter entièrement sur un seul processeur. Dans cette classe, l'approche hybride (Section 2.3.4.2) est utilisée en affectant d'abord les *jobs* aux différentes files, et au niveau de chacune des files, les *jobs* sont ordonnancés localement.

Pleine migration : Cette classe n'impose aucune restriction quant à la migration des *jobs*.

Dans cette thèse, nous n'autorisons pas la migration des tâches et chaque cœur reçoit un sous-ensemble de tâches après l'application de l'heuristique de partitionnement.

2.3.2.4 Dépendance entre tâches

Une dépendance entre deux tâches τ_i et τ_j peut être de deux types : une dépendance de précédence et/ou une dépendance de données. Une dépendance de précédence entre (τ_i, τ_j) impose que la tâche τ_j commence son exécution après que la tâche τ_i ait complètement fini de s'exécuter [48–51]. Les contraintes de précédences sont indirectement des contraintes temps-réel et on dit que la tâche τ_i est un prédécesseur de la tâche τ_j et τ_j est un successeur de τ_i. Si τ_i s'exécute exactement une fois avant une exécution de τ_j on a une contrainte de précédence simple sinon une contrainte de précédence étendue [52–54]. De plus, entre chaque paire de tâches périodiques dépendantes, un message périodique peut être échangé entre elles sur le médium de communication, il s'agit donc de la dépendance de données. Après un événement de reconfiguration, un ou plusieurs messages peuvent être ajoutés sur le médium de communication et cela peut augmenter le trafic sur le support de communication et ne pas satisfaire les échéances des messages.

Nous attaquons la problématique de communication entre tâches dans le chapitre traitant les architectures multi-cœurs. Nous ajoutons dans ce chapitre une contrainte de communication qui vérifie que chaque médium de communication est faisable et que tous les messages peuvent arriver à leur destination en satisfaisant la contrainte temporelle.

2.3.3 Ordonnancement temps-réel mono-cœur

Dans le cadre simple de l'ordonnancement mono-cœur où toutes les tâches sont considérées indépendantes, il existe deux types d'assignations de priorités [5] :

1. priorités fixes : aussi appelée assignation à priorité statique, où chaque tâche reçoit une priorité par rapport aux autres à l'initialisation du système et chaque *job* lancé hérite de la priorité de la tâche de laquelle il découle ;
2. priorités dynamiques : la priorité est donnée aux *jobs* des tâches, et non pas aux tâches. Selon la règle d'assignation utilisée, la priorité évolue au cours du temps tout au long de l'exécution du système.

Nous présentons ci-après les algorithmes d'ordonnancement les plus souvent rencontrés dans la littérature.

2.3.3.1 Politiques d'ordonnancement à priorités fixes

Une priorité fixe aux tâches est une priorité qui ne varie pas au cours de l'exécution de la tâche. Les algorithmes d'ordonnancement à priorités fixes les plus utilisés sont "Rate Monotonic" et "Deadline Monotonic".

Rate Monotonic :
L'algorithme Rate Monotonic (RM) a été introduit par Liu and Layland en 1973 [40]. C'est un algorithme d'ordonnancement préemptif qui s'applique à des tâches périodiques indépendantes, et à échéance sur requête ($T_i = D_i$). La priorité d'une tâche est inversement proportionelle à sa période, c'est-à-dire que plus la période d'une tâche est petite, plus sa priorité est grande. Cet algorithme est optimal dans la classe des algorithmes à priorités fixes pour les tâches indépendantes préemptibles à échéance sur requête non concrètes ou synchrones. Une condition suffisante d'ordonnançabilité de l'algorithme

d'ordonnancement RM [40] pour un ensemble de N tâches périodiques Π à échéance sur requête est donnée par :

$$\sum_{i=1}^{N} \frac{W_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \tag{2.1}$$

Deadline Monotonic :

Bien que RM soit optimal dans la classe des tâches périodiques à échéances sur requêtes, ce n'est plus le cas lorsque les tâches sont à échéances contraintes ($D_i \leq T_i$). On retrouve donc dans [55], un algorithme proposé par Leung et Whitehead connu sous le nom de Deadline Monotonic (DM) prenant en compte les délais critiques D_i de chaque tâche τ_i tout en gardant le principe d'ordonnancement à priorités fixes. La priorité d'une tâche est inversement proportionelle à son échéance relative, c'est-à-dire que plus l'échéance relative d'une tâche est petite, plus sa priorité est grande. Cet algorithme est optimal dans la classe des algorithmes préemptifs à priorités fixes pour les tâches indépendantes préemptibles à échéances contraintes. Un ensemble de N tâches périodiques est ordonnançable par DM si :

$$\sum_{i=1}^{N} \frac{W_i}{D_i} \leq N(2^{\frac{1}{N}} - 1) \tag{2.2}$$

2.3.3.2 Politiques d'ordonnancement à priorités dynamiques

Les ordonnanceurs à priorités fixes ont, par leur simplicité de mise en œuvre, servi de base dans l'ordonnancement. Les algorithmes à priorités dynamiques sont quant à eux plus difficiles à mettre en œuvre mais offrent souvent de meilleures performances. Les priorités cette fois-ci évoluent en fonction du temps. Nous présentons ci-après deux algorithmes dont l'un est fondé sur l'évaluation des échéances absolues des *jobs* et l'autre sur la laxité des tâches.

Earliest Deadline First :

L'algorithme Earliest Deadline First (EDF), a été présenté par Jackson en 1955 [56] puis par Liu and Layland [40]. EDF est un algorithme d'ordonnancement qui peut être préemptif ou non préemptif et qui s'applique à des tâches périodiques indépendantes à échéance sur requête ($T_i = D_i$). La plus grande priorité à la date t est allouée à la tâche dont l'échéance absolue est la plus proche. EDF est optimal pour les tâches indépendantes

préemptives. Selon [40], un ensemble Π de N tâches périodiques à échéances sur requêtes est ordonnançable par EDF si et seulement si le facteur d'utilisation U du processeur vérifie la condition suivante :

$$\sum_{i=1}^{N} \frac{W_i}{T_i} \leq 1 \tag{2.3}$$

EDF est certainement l'algorithme le plus populaire. En effet, EDF est un algorithme très performant qui permet d'atteindre un facteur d'utilisation du processeur de 100% sous la condition énoncée dans l'Eq. 2.3. De plus ce résultat est aussi valide pour des ensembles de tâches à départs différés, ce qui illustre bien la supériorité de EDF sur RM ou DM.

Least-Laxity First :

L'algorithme d'ordonnancement Least-Laxity First (LLF), présenté par Mok et Dertouzos [57], se base sur la laxité qui représente le temps restant avant l'occurence de sa date de démarrage ou de reprise au plus tard. La tâche dont la laxité est la plus faible comparée à toutes les tâches prêtes aura la plus grande priorité. Cet algorithme est optimal pour les tâches indépendantes préemptives. D'après [58], la condition d'ordonnançabilité de LLF est la même que pour EDF, c'est-à-dire que la condition d'ordonnançabilité nécessaire et suffisante de LLF pour un ensemble de tâches périodiques à échéance sur requête est donnée par :

$$\sum_{i=1}^{N} \frac{W_i}{T_i} \leq 1 \tag{2.4}$$

Bien qu'il soit aussi performant que EDF, LLF engendre cependant un nombre supérieur de préemptions donc de changements de contexte, ce qui explique qu'il soit aussi peu utilisé dans le cas monoprocesseur [59]. De plus, Georges et al. ont prouvé que LLF n'est plus optimal dans le cas d'un ordonnancement non-préemptif [60].

Dans nos travaux, nous avons choisi l'algorithme EDF comme algorithme d'ordonnancement vues son optimalité et ses performances.

2.3.3.3 Stratégies d'ordonnancement du point de vue énergétique

Il devient important de s'intéresser à la conception de politiques de gestion de l'énergie pour concevoir des systèmes entièrement autonomes. Du point de vue recherche, une thématique intéressante repose sur l'étude de politiques d'ordonnancement de tâches temps-réel sous contraintes énergétiques. Il existe une littérature pour ce type de problèmes [61–63]. Les techniques existantes dans le domaine de la gestion d'énergie des systèmes temps-réel embarqués et récupérant l'énergie de l'environnement s'appuient principalement sur des techniques basées sur la variation de la vitesse de fonctionnement du processeur appelées DVFS (*Dynamic Voltage Frequency Scaling*). Plusieurs études basées sur les processeurs supportant la technique DVFS, suggèrent la modification temporelle des paramètres des tâches afin de développer des systèmes temps-réel reconfigurables [2, 16, 18, 64]. La recherche de Wang et al. dans [2] met l'accent sur la reconfiguration temps-réel des systèmes embarqués sous les contraintes d'énergie. Les auteurs économisent de l'énergie en ralentissant les tâches lorsque l'énergie stockée n'est pas suffisante. Leur approche propose un agent en vue de réduire la consommation d'énergie après l'application d'un scénario de reconfiguration. Vu que la consommation de puissance est proportionnelle à l'utilisation du processeur, Wang et al. dans [2] proposent une stratégie d'ordonnancement qui rend possible l'exécution d'une reconfiguration dynamique des systèmes temps-réel où l'ajout et la suppression de tâches sont appliqués au moment de l'exécution. Cette approche vise à minimiser la consommation énergétique du système par l'allongement des périodes T_i des tâches par l'attribution d'une période unique à toutes les tâches. Une autre solution proposée est de réduire les WCETs W_i par l'attribution d'une WCET unique pour toutes les tâches. Différentes solutions sont également proposées pour les modifications de WCETs, les délais et les périodes de tâches [7, 31, 37, 38]. Bien que ces solutions peuvent baisser la consommation énergétique, il n'est pas raisonnable pour un système temps-réel de modifier la période des tâches au delà d'une certaine limite. De plus, si les tâches ont des périodes très diverses, les tâches qui ont des petites périodes sont fortement affectées si elles sont alignées sur les tâches ayant de grandes périodes. Notre problématique porte principalement sur la reconfiguration des systèmes temps-réel lorsque des reconfigurations dynamiques des tâches périodiques sont appliquées au moment de l'exécution. De plus, nous vérifions après chaque reconfiguration que le système satisfait ses contraintes, notamment temps-réel, énergétiques,

etc. Nos solutions proposées dans cette thèse sont basées sur l'adaptation en ligne des paramètres T_i et W_i en utilisant une classification des tâches en packs.

2.3.4 Ordonnacement temps-réel multi-cœurs

Cette partie s'adresse à l'ordonnancement de tâches sur des architectures multi-cœurs. Nous commencerons tout d'abord par classifier les différentes architectures multi-cœurs. Par la suite, nous présentons les différentes approches d'ordonnancement.

2.3.4.1 Classification des architectures multi-cœurs

Une architecture multi-cœurs est composée de plusieurs cœurs, et plusieurs tâches peuvent s'exécuter en parallèle sur les différents cœurs. Nous distinguons trois types d'architectures multi-cœurs [65] selon qu'elles soient constituées de :

- cœurs identiques : cœurs qui ont la même capacité de calcul s (nombre d'instructions traitées par seconde) ;
- cœurs uniformes : Chaque cœur C_k d'une architecture uniforme est caractérisé par sa capacité de calcul s_k. Lorsqu'un *job* s'exécute sur un cœur C_k de capacité de calcul s_k pendant t unités de temps, il réalise $s_k * t$ instructions sur ce cœur ;
- cœurs indépendants : Pour ce type d'architecture, la capacité de calcul dépend non seulement du cœur mais également de la tâche qui s'exécute. Ainsi, on définit une capacité de calcul $s_{i,k}$ associée à chaque couple tâche-cœur (τ_i, C_k) de telle sorte que la tâche τ_i réalise $s_{i,k} * t$ instructions lorsqu'elle s'exécute sur le cœur C_k pendant t unités de temps.

Les architectures identiques correspondent à des architectures homogènes alors que les architectures uniformes et indépendantes correspondent à des architectures hétérogènes.

Nous nous placerons dans le cadre d'une architecture à p cœurs hétérogènes où p représente le nombre de cœurs qui composent l'architecture.

2.3.4.2 Approches d'ordonnancement multi-cœurs

Un algorithme d'ordonnancement multi-cœurs détermine pour chaque tâche, le cœur sur lequel cette tâche doit s'exécuter (problème de placement) et sur chaque cœur, la date et l'ordre de démarrage d'exécution des tâches (problème d'ordonnancement). En effet, l'ordonnancement multi-cœurs est un problème à 2 dimensions : i) l'organisation spatiale (placement des tâches sur les cœurs) et ii) l'organisation temporelle des tâches (stratégie d'ordonnancement) [65]. Il existe deux types fondamentaux de stratégies multi-cœurs : les approches globales et les approches par partitionnement.

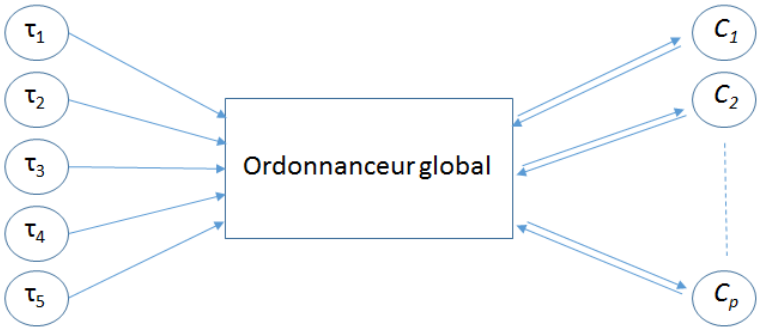

FIGURE 2.5: Représentation graphique de la stratégie d'ordonnancement global.

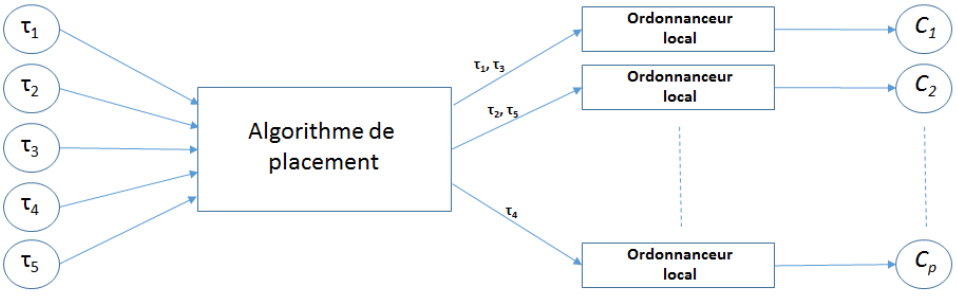

FIGURE 2.6: Représentation graphique de la stratégie d'ordonnancement par partitionnement.

Il existe aussi une stratégie d'ordonnancement hybride dite de semi-partitionnement obtenue par combinaison de la stratégie globale et celle par partitionnement.

Avant de définir le comportement adopté lors de l'utilisation de chacune de ces stratégies, nous considérons un ensemble de N tâches temps-réel indépendantes préemptibles à échéances sur requêtes noté Π et une architecture multi-cœurs composée de p cœurs notée Γ.

Approche globale

L'objectif est d'attribuer, à chaque instant, les N tâches aux p cœurs. Pour cela, il existe une seule file d'attente pour l'ensemble des tâches et on utilise une stratégie d'ordonnancement unique qui s'applique sur l'ensemble des cœurs (voir Figure 2.5). Dans ce cas, une propriété essentielle est que la migration des tâches soit autorisée.

Approche par partitionnement

L'objectif est d'affecter définitivement chaque tâche à un cœur, autrement dit, de définir p sous-ensembles de tâches attribués chacun à un cœur. L'objectif réside dans le fait qu'à chaque cœur C_k soit associé un sous-ensemble de tâches tel que l'intersection de chaque ensemble soit vide et que l'union des ensembles soit l'ensemble de tâches Π. Notons que la migration de tâches n'est pas autorisée. De ce fait, le problème multi-cœurs à p cœurs se résume alors à p problèmes mono-cœur. Il est ainsi possible d'appliquer des stratégies mono-cœur sur chacun des processeurs de l'architecture, qui possèdent chacun leur propre file de *jobs* en attente d'exécution (voir Figure 2.6).

Approche hybride

L'approche hybride ou semi-partitionnée est dérivée de l'approche partitionnée. Dans cette approche, certains *jobs* d'une tâche peuvent être exécutés sur des cœurs différents alors que certaines tâches ne sont pas du tout autorisées à migrer, ce qui entraîne des migrations moins nombreuses.

Il faut noter que pour toutes ces approches, il existe une condition nécessaire d'ordonnançabilité.

Condition nécessaire : D'après [66], un ensemble de N tâches périodiques sur p cœurs est ordonnançable si et seulement si :

$$U = \sum_{i=1}^{N} \frac{W_i}{T_i} \leq p \tag{2.5}$$

Dans nos travaux, nous avons opté pour le développement de stratégies basées sur l'approche par partitionnement du fait que cette dernière réduit le problème multi-cœurs à p problèmes mono-cœur. Cette approche rend ainsi possible l'application de stratégies mono-cœur sur chacun des cœurs de l'architecture possédant chacun leur propre file de

jobs en attente d'exécution. C'est vrai que ce choix ne donne pas toujours un ordonnancement optimal, mais il réduit l'overhead de la stratégie puisque les tâches sont affectées définitivement dès le premier placement.

2.3.4.3 Heuristiques de placement de tâches pour minimiser la consommation énergétique

Les termes placement, allocation, partitionnement ou assignation des tâches sont équivalents dans le vocabulaire de l'ordonnacement temps-réel multi-cœurs [22, 67, 68]. Dans la suite nous utiliserons le terme placement. Placer une tâche sur un cœur signifie l'exécuter sur un cœur de façon à ce que cette tâche, et toutes les autres tâches qui sont déjà placées sur ce même cœur, soient ordonnançables selon une condition d'ordonnançabilité qui est précisée. Dans cette partie, nous étudions quelques approches existantes dans la résolution du problème de placement en minimisant le coût de communication.

Dans la littérature, certains travaux [69–72] considèrent que la consommation énergétique des systèmes multi-cœurs hétérogènes est fortement dépendante du placement des tâches. Huang et al. [69, 70] proposent des heuristiques de placement des tâches pour minimiser la consommation énergétique. Ils supposent que la consommation énergétique est proportionnelle à la distance entre les cœurs. Autrement dit, plus les tâches dépendantes sont placées sur des cœurs différents, plus la consommation d'énergie augmente. Par conséquent, minimiser le coût de communication entre les cœurs devient un challenge pour augmenter les performances des systèmes. En effet, pour réduire la consommation énergétique, il faut trouver un placement de tâches qui minimise le coût de communication. Dans ce contexte, Kapil Govil propose dans [21] un algorithme de placement pour placer m tâches sur n processeurs ($m > n$). L'algorithme est basé sur *Task Clustering* en groupant chaque ensemble de tâches communicantes dans le même *Cluster* et ensuite l'algorithme propose si c'est possible de placer chaque *Cluster* dans le même processeur afin de minimiser le coût global de communication entre les processeurs. Cet algorithme a été amélioré par Bhardwa et al. en 2013 [22] en proposant une heuristique basée sur le même principe (celui dans [21]) mais en ajoutant une contrainte de vérification sur les charges des processeurs. Bien que cette heuristique de placement de tâches permet de réduire le coût global de communication, elle donne aussi un placement où la charge des processeurs est équilibrée. L'équilibrage de charge des processeurs est un paramètre

supplémentaire proposé dans [22] pour améliorer la performance du système. Bhardwa et al. [22] ont évalué leur heuristique et ils ont démontré qu'elle est plus performante que l'algorithme présenté dans [21].

Il existe d'autres travaux récemment publiés [17, 19, 30] qui traitent la problématique de minimisation de la consommation énergétique d'un MPSoC en adaptant les paramètres des messages échangés entres les processeurs. Ces approches sont applicables lors de l'exécution du système. Afin de minimiser la consommation énergétique, Khemaissia et al. [19, 30] proposent une approche qui ralentit les messages échangés sur le médium de communication lorsque l'énergie stockée n'est pas suffisante. Une autre alternative proposée par les auteurs pour minimiser la consommation énergétique est de supprimer les messages moins prioritaires. Cette approche permet de baisser le taux d'utilisation sur le NoC et, par conséquent, la consommation énergétique est diminuée.

À notre connaissance, aucun travail n'aborde le problème de placement des tâches dépendantes sur les architectures multi-cœurs hétérogènes reconfigurables. À l'heure actuelle, aucune étude scientifique n'a considéré l'adaptation des paramètres des tâches lors de leur placement comme solution pour minimiser le coût global de communication.

2.4 Problèmes d'optimisation et méthodes de résolution pour la problématique de placement de tâches

Dans cette partie, nous étudions quelques méthodes les plus utilisées dans la résolution des problèmes de placement des tâches. Nous distinguons deux grandes catégories de méthodes de résolution : celles qui sont dites exactes et celles approchées.

2.4.1 Méthodes exactes ou optimales

Une méthode exacte est une méthode qui permet de trouver une solution faisable en parcourant toutes les combinaisons possibles de placement de tâches, si celle-ci existe. Par ailleurs si cette méthode ne trouve pas de solutions, cela signifie que le problème n'a pas de solution. Et puisque le problème de placement inclut une optimisation, la solution trouvée par une méthode exacte est la meilleure solution possible dite optimale pour le critère défini [73].

2.4.1.1 Branch and bound

La méthode du Branch and Bound (B&B) (procédure par évaluation et séparation progressive) [74–76] consiste à construire un arbre d'exploration qui énumère toutes les solutions. Ensuite, en utilisant certaines propriétés du problème en question, il faut trouver une manière intelligente d'explorer cet arbre. Cette méthode utilise une fonction qui permet de fixer une ou plusieurs bornes (tout dépend du problème traité) afin d'évaluer certaines solutions pour, soit les exclure, soit les maintenir comme des solutions potentielles. Bien entendu la performance d'une méthode de B&B dépend, entre autres, de la qualité de cette fonction (de sa capacité d'exclure des solutions partielles le plus tôt possible). La méthode de B&B est utilisée dans de nombreux travaux de placement et d'ordonnancement temps-réel multiprocesseurs comme dans [51, 75, 77]. Chen et al. [75] ont utilisé le B&B pour minimiser le *makespan* en prenant en compte les coûts de communications entre les tâches non-préemptives. Le modèle d'architecture pris en compte est une architecture hétérogène où tous les processeurs peuvent communiquer les uns avec les autres à travers les médiums de communication. Cependant ces algorithmes ne garantissent pas le respect des contraintes temps-réel. Peng et al. [77] appliquent le principe du B&B au problème de placement de tâches périodiques dépendantes. Les travaux dans [51] proposent un algorithme optimal du type B&B d'ordonnancement de tâches sous contraintes de précédence.

2.4.1.2 Programmation linéaire en nombre entiers

La programmation linéaire (ou *linear programming* LP) [78, 79] est une méthode d'optimisation mathématique dans lequel nous cherchons à minimiser (ou maximiser) une fonction objectif exprimée à l'aide de variables de décisions. Les solutions à trouver doivent être représentées par des variables réelles. S'il est nécessaire d'utiliser des variables discrètes dans la modélisation, on parle de programmation linéaire en nombres entiers (ou *integer linear programming* ILP). Un problème d'ILP est un problème de LP dans lequel on ajoute la contrainte supplémentaire que certaines variables ne peuvent prendre que des valeurs entières.

La forme standard d'un problème ILP est :

$$\begin{cases} \text{Minimiser/Maximiser } c^T x \\ \text{tel que} \\ \quad Ax + s = b \\ \quad s \geq 0 \\ \quad x \in \mathbb{Z} \end{cases}$$

Avec :

- b et c sont des vecteurs et A une matrice de valeurs entières.
- x et s sont des variables à rechercher.

Une fois le problème correctement formulé, la résolution peut être confiée à des solveurs. Les solveurs sont des programmes qui appliquent des méthodes de manière à trouver des solutions optimales au problème d'optimisation linéaire formulé. Nous citons comme exemple de solveurs : lpsolve [80] et CPLEX [81]

Dans ce contexte, les problèmes de placement et d'ordonnancement de tâches sont, en général, des problèmes d'optimisation puisque leur objectif est de minimiser (ou maximiser) une fonction objectif en respectant certaines contraintes. Résoudre un problème d'optimisation consiste à trouver la (ou les) meilleure(s) solution(s), vérifiant un ensemble de contraintes et d'objectifs définis par l'utilisateur [82]. Le placement d'une tâche sur un cœur se faisant en fonction d'un critère, le problème à résoudre est un problème d'optimisation. Le critère d'optimisation peut être le coût de communication entre cœurs, le nombre de cœurs, le facteur d'utilisation, etc. Certains travaux [83, 84] utilisent la formulation ILP pour formaliser les problèmes de placement de tâches. Chatterjee et al. [83] proposent une stratégie de placement et d'ordonnancement de tâches pour minimiser la consommation énergétique du système. Le principe de l'algorithme est de placer les tâches qui communiquent fréquemment dans la même ressource de calcul. Cette stratégie donne de meilleures performances en termes de consommation d'énergie par rapport aux autres algorithmes. Tousun et al. [84] présentent une formulation ILP de placement des tâches dépendantes sur une architecture multi-cœurs dont les cœurs sont connectés par un NoC. La fonction objectif de la formulation est de minimiser la consommation énergétique consommée par un NoC Mesh.

Dans nos travaux, nous formaliserons tous les problèmes en utilisant la formulation ILP afin de pouvoir produire des résultats optimaux. L'idée est de pouvoir situer nos solutions proposées par rapport aux solutions optimales produites par un solveur. Les résolutions ont été réalisées avec le solveur CPLEX [81].

2.4.2 Méthodes approchées ou sous-optimales

Les méthodes exactes, bien que produisant des solutions optimales ne sont pas généralement applicables à cause de leur temps de calcul énorme pour des problèmes réalistes. Les méthodes approchées (ou sous-optimales) permettent de donner des solutions qui dans la plupart des cas ne sont pas optimales, mais ayant des temps de calcul acceptables qui les rendent applicables dans les problèmes de grandes tailles.

2.4.2.1 Heuristiques de listes

Généralement ces heuristiques utilisent un graphe acyclique orienté (DAG) pour représenter les dépendances entres les tâches en associant des poids aux arcs (symbolisant les coûts de communication entre processeurs) qui relient les sommets (représentant les tâches), ainsi que des poids aux sommets (symbolisant les durées d'exécution des tâches). Les heuristiques de liste [85–87] définissent une liste de tâches en attribuant à chacune d'elles une priorité. Cette liste détermine l'ordre du choix de placement d'une tâche sur les différents processeurs. À chaque étape de l'heuristique, une tâche est sélectionnée dans la liste des tâches candidates pour être ordonnancée sur un processeur puis la liste est mise à jour (suppression de la tâche ordonnancée et ajout des tâches qui n'ont pas de prédécesseurs dans le graphe des tâches). Ceci est répété jusqu'à ce que la liste soit vide.

2.4.2.2 Heuristiques de regroupement ou *Clustering*

Les heuristiques de regroupement (ou *Clustering*) [21, 22, 88–91] consistent à regrouper les tâches en différents sous-ensembles appelés *Clusters* en fonction des objectifs du problème (minimisation des coûts de communication, respect des contraintes temps-réel, etc.). Les tâches d'un *Cluster* sont ensuite placées sur le même processeur, ce qui diminue la taille du problème. La résolution du problème de placement par des heuristiques de

regroupement est donc décomposée en deux parties. La première partie constitue le regroupement proprement dit et la seconde consiste à placer les tâches sur les processeurs en fonctions des contraintes temps-réel, communication, etc.

Nous nous intéressons dans cette thèse aux heuristiques de placement par regroupement des tâches communiquantes, Clustering, notamment celles proposées dans [21, 22].

2.5 Conclusion

Dans ce chapitre, nous avons présenté dans une première étape l'architecture matérielle cible afin d'illustrer l'impact d'ajout de tâches sur les contraintes, notamment temps-réel, énergétiques et communications. Par la suite, nous avons évoqué les différents travaux traitant la problématique d'ordonnancement sur ce type d'architecture. Une critique de ces travaux a été faite afin de positionner nos contributions qui s'intéressent à la reconfiguration des systèmes reconfigurables mono et multi-cœurs sous contraintes énergétiques. Enfin, nous avons présenté des méthodes d'optimisation et de formalisation du problème de placement de tâches. Les chapitres à venir détailleront nos contributions qui sont dédiées aux architectures mono et multi-cœurs (homogènes et hétérogènes).

Chapitre 3

Contribution à l'ordonnancement temps-réel sous contrainte d'énergie pour les architectures mono-cœur

3.1 Introduction

Parmi les solutions citées dans l'état de l'art pour tenter de résoudre la problématique de l'ordonnancement sous contrainte énergétique, l'une d'elles repose sur la modification des paramètres des tâches [2]. Le modèle sous-jacent est basé sur la théorie d'élasticité des tâches [14]. C'est dans ce contexte que notre travail s'inscrit. Notre objectif consiste à développer une nouvelle stratégie d'ordonnancement permettant le respect des contraintes temps-réel et énergétique. Cette stratégie peut être déployée sur des systèmes embarqués reconfigurables et rechargeables périodiquement. Avant de détailler les différentes étapes de la stratégie ainsi que les solutions proposées, nous débutons ce chapitre par une formalisation d'un système temps-réel embarqué reconfigurable et rechargeable $RTSys$ en décrivant son modèle de tâches ainsi que son modèle de consommation énergétique. Par la suite, nous formaliserons mathématiquement la fonction objectif à optimiser ainsi que les contraintes temps-réel et énergétique et nous détaillerons ensuite les heuristiques proposées qui supporteront la stratégie d'ordonnancement. Nous finirons par présenter différents scénarios de simulation pour montrer que notre approche permet d'améliorer les résultats de l'état de l'art sur ce sujet. Notre approche sera également comparée avec

la solution optimale que nous pouvons obtenir par la résolution de la formulation ILP du problème.

3.2 Formalisation du système basé sur l'architecture mono-cœur

Dans cette section, seront représentés les modèles de tâches, de consommation d'énergie et d'utilisation du processeur afin de formaliser le problème. Nous considérons dans ce chapitre que la plate-forme est constituée d'un système mono-cœur supportant une application temps-réel reconfigurable, noté $RTSys$. L'application est supposée être composée de N tâches périodiques indépendantes.

3.2.1 Modèle de tâches

Le modèle de tâches périodiques classique, dit de Liu et Layland [40], est le plus utilisé dans la modélisation des systèmes temps-réel. Nous supposons que $RTSys$ exécute un ensemble de tâches périodiques indépendantes Π tel que : $\Pi = \{\tau_1, \tau_2, ..., \tau_N\}$. Dans cet ensemble Π, certaines tâches sont considérées comme étant "flexibles" [14, 45], ce qui signifie que leur période peut être modifiée dans une plage spécifiée. Selon [7, 14, 45], une tâche périodique τ_i est modélisée par un ensemble de paramètres comme suit : τ_i=$\{R_i, W_i, T_i, T_{i_{max}}, D_i, I_i\}$, avec :

- Temps de début R_i : C'est le temps de début de l'exécution de la tâche τ_i,
- WCET W_i : Le pire temps d'exécution de la tâche τ_i,
- Une période T_i : L'intervalle régulier d'arrivée de la tâche τ_i,
- Une période max $T_{i_{max}}$: C'est la période maximale que τ_i peut accepter,
- Échéance D_i : Le temps limite absolu pour l'exécution de la tâche τ_i,
- Facteur d'importance I_i : C'est un facteur définit par le concepteur qui est accordé à chaque τ_i. Ce paramètre désigne le niveau d'importance de la tâche. La tâche τ_i la plus importante est celle qui a la valeur I_i la plus petite.

Nous supposons que chaque tâche τ_i est préemptible et à échéance sur requête ($T_i = D_i$). Nous supposons que le cœur ordonnance les tâches en utilisant l'algorithme EDF [40].

3.2.2 Modèle de facteur d'utilisation du processeur/cœur

Selon Liu et Layland [40], le taux d'utilisation du processeur par une tâche τ_i est :

$$u_{\tau_i} = \frac{W_i}{T_i} \tag{3.1}$$

Ainsi, le facteur d'utilisation (ou l'utilisation du processeur) est le pourcentage de temps que le processeur passe pour exécuter l'ensemble des N tâches du système, et vaut :

$$U_C = \sum_{i=1}^{N} u_{\tau_i} = \sum_{i=1}^{N} \frac{W_i}{T_i} \tag{3.2}$$

Un ordonançement sous EDF est possible sur un cœur C si et seulement si :

$$U_C \leq 1 \tag{3.3}$$

Eq. 3.3 est la contrainte temps-réel, notée dans la suite du rapport par **RTConst**.

3.2.3 Modèle de consommation d'énergie

Comme nous l'avons défini dans la Section 2.3.3.3, la majorité des processeurs s'appuie principalement sur des techniques basées sur la variation de la vitesse de fonctionnement du processeur appelées $DVFS$. En effet, la baisse de la fréquence d'horloge du processeur permet de réduire la tension d'alimentation. Nous supposons dans ce travail, que nous considérons une architecture mono-cœur supportant la technique $DVFS$, avec F_n est la fréquence nominale du cœur qui est dans l'ensemble $[F_1 \dots F_{max}]$. Les valeurs dans cet ensemble correspondent aux points de fonctionnement du cœur.

Par ailleurs, dans la littérature [2, 15], la puissance électrique consommée dans les circuits réalisés avec la technologie CMOS est souvent divisée en deux composantes : une composante dynamique due à l'activité des transistors et une composante statique due à leurs courants de fuite. En se basant sur la démonstration effectuée par Wang et al. dans [15, 92], nous constatons que la puissance consommée P du système est proportionnelle

au carré de l'utilisation du processeur U, i.e, $P = k * U^2$.

Afin de décrire le modèle d'énergie du système, nous supposons que $RTSys$ est alimenté par une batterie rechargeable périodiquement. Le modèle d'énergie de cette batterie est caractérisé par :

- $E(t)$: L'énergie disponible dans la batterie à l'instant t.
- $t_{recharge}$: Le temps restant pour arriver à la prochaine recharge.

Et comme $P = k * U^2$, donc $P = k * \left(\sum_{i=1}^{N} \frac{W_i}{T_i}\right)^2$.

Afin de simplifier le calcul, nous supposons que $k = 1$, on obtient donc, $P = \left(\sum_{i=1}^{N} \frac{W_i}{T_i}\right)^2$.

À un instant donné t et pour garantir que le système peut s'exécuter correctement jusqu'à la prochaine recharge en gardant la puissance consommée actuelle, il est nécessaire de vérifier que :

$$P(t) * t_{recharge} \leq E(t) \tag{3.4}$$

Avec $P(t)$ la puissance consommée à l'instant t. Par ailleurs, nous définissons la variable $P_{Limit}(t)$ par la quantité $\frac{E(t)}{t_{recharge}}$, i.e,

$$P_{Limit}(t) = \frac{E(t)}{t_{recharge}} \tag{3.5}$$

C'est la consommation maximale tolérable à l'instant t, sinon $RTSys$ ne respecte pas sa contrainte d'énergie et il ne sera plus capable de s'exécuter jusqu'à la prochaine recharge. Pour conclure, la satisfaction de la contrainte énergétique du système est vérifiée par :

$$P(t) \leq P_{Limit}(t) \tag{3.6}$$

Eq. 3.6 est la contrainte énergétique, notée dans la suite du rapport par **EnergyConst**.

3.3 Problème de reconfiguration

Nous supposons que $RTSys$ est initialement composé par $\Pi(t_0) = \{\tau_1, \tau_2, ..., \tau_{n_1}\}$. Nous supposons que le système est initialement faisable. Un système est dit faisable si et seulement si, il satisfait les contraintes temps-réel et énergétique. Nous supposons que $RTSys$

se reconfigure dynamiquement à l'instant t_i en ajoutant n_2 tâches. La nouvelle implémentation devient alors comme suit : $\Pi(t_i) = \{\tau_1, \tau_2, ..., \tau_{n_1}, \tau_{n_1+1}, ..., \tau_N\}$, avec $N = n_1 + n_2$. Ce scénario de reconfiguration est nommé *RE-CONF*. Après l'ajout de plusieurs tâches, l'utilisation du processeur augmente et *RTSys* risque de devenir non faisable à cause de la violation de l'une de ses contraintes. L'objectif de ce chapitre est d'assurer la faisabilité de *RTSys* après un ou plusieurs scénario(s) de reconfiguration. Cette section expose la modélisation par programmation linéaire du problème à résoudre.

3.3.1 Étude de cas : *RE-CONF*

Nous supposons que *RTSys* possède un cœur C exécutant initialement quatre tâches indépendantes avec la politique d'ordonnancement EDF, $\Pi(t_0) = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. Les caractéristiques des tâches sont illustrées dans le Tableau 3.1.

TABLE 3.1: Tâches initiales.

Task	W_i	T_i	$T_{i_{max}}$	D_i
τ_1	4	40	100	40
τ_2	6	15	85	15
τ_3	3	29	99	29
τ_4	4	40	100	40

L'utilisation du cœur est $U_C = \sum_{i=1}^{4} \frac{W_i}{T_i} = \frac{4}{40} + \frac{6}{15} + \frac{3}{29} + \frac{4}{40} = 0.7 \leq 1$. *RTSys* est donc ordonnançable. En supposant que k=1, la puissance consommée est $P(t_0) = \left(\sum_{i=1}^{4} \frac{W_i}{T_i}\right)^2 = 0.7^2 = 0.49$.

À l'instant t_i, *RTSys* subit un scénario de reconfiguration qui ajoute deux autres tâches τ_5 et τ_6 (Tableau 3.2), l'ensemble devient $\Pi(t_i) = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6\}$. Après cet événement, *RTSys* devient non ordonnançable puisque la nouvelle valeur de $U_C = \sum_{i=1}^{6} \frac{W_i}{T_i} = \frac{4}{40} + \frac{6}{15} + \frac{3}{29} + \frac{4}{40} + \frac{5}{20} + \frac{6}{25} = 1.19 > 1$. La contrainte temps-réel n'est pas donc respectée.

TABLE 3.2: Tâches ajoutées.

Task	W_i	T_i	$T_{i_{max}}$	D_i
τ_5	5	20	90	20
τ_6	6	25	105	25

La nouvelle valeur de puissance consommée est $P(t_i) = 1.19^2 = 1.41$.

Pour vérifier que la contrainte énergétique n'a pas été violée après ce scénario, il faudrait que la valeur de $P_{Limit}(t_i)$ (donnée) soit supérieure ou égale à $P(t_i) = 1.41$.
Afin de vérifier cette contrainte, nous supposons qu'à l'instant t_i :

- La quantité d'énergie restante dans la batterie est : $E(t_i) = 2502$
- Le temps restant jusqu'à la prochaine recharge est : $t_{recharge} = 1800$

En appliquant l'Eq. 3.6, nous obtenons $P_{Limit}(t_i) = \frac{2502}{1800} = 1.39$.
Puisque $P_{Limit}(t_i) < P(t_i) = 1.41$, la contrainte énergétique est donc violée.

3.3.2 Problème de contraintes temps-réel

Après le scénario *RE-CONF*, si le cœur C n'arrive pas à ordonnancer ses tâches ($U_C > 1$), nous proposons de modifier le paramètre de la période de façon à réduire le taux d'utilisation U_C. En modifiant ce paramètre, l'objectif est de faire descendre le taux d'utilisation à une valeur inférieure ou égale à 1. Le principe global de l'approche proposée est le suivant :

1. Nous allongeons les périodes des tâches en incrémentant progressivement la période de chaque tâche jusqu'à arriver à un taux d'utilisation du processeur U_C inférieur ou égal à 1, mais cela ne sera possible que jusqu'à une certaine valeur maximale $T_{i_{max}}$.
2. Si U_C est encore supérieur à 1, nous proposons d'augmenter la fréquence du cœur, ce qui permet de réduire le temps d'exécution (W_i) de certaines tâches.
3. Si U_C est encore supérieur à 1, nous supprimons certaines tâches selon leur facteur d'importance I_i.

Afin d'exploiter au maximum possible l'allongement des périodes des tâches et de minimiser la modification de la fréquence d'horloge, nous modélisons les deux premières étapes de cette approche par une formulation ILP. Si la résolution de ce système ne retrouve pas l'ordonnançabilité du système, la troisième étape est appliquée pour supprimer certaines

tâches afin de rendre le système faisable.

$$Pb1 = \begin{cases} \textbf{Minimiser} \sum_{i=1}^{N} \Delta W_i \\ t.q. \\ \quad \sum_{i=1}^{N} \frac{W_i - \Delta W_i}{T_i + \Delta T_i} \leq 1 \\ \quad W_i - \Delta W_i > 0, \ \forall \ i \in [1 \dots N] \\ \quad T_i + \Delta T_i \leq T_{i_{max}}, \ \forall \ i \in [1 \dots N] \\ \quad F_n * W_i/(W_i - \Delta W_i) \geq F_1 \\ \quad F_n * W_i/(W_i - \Delta W_i) \leq F_{max} \end{cases}$$

Avec, ΔT_i est un entier à ajouter à la période de tâche τ_i afin de baisser le taux d'utilisation du cœur C. ΔW_i représente la valeur à soustraire au temps d'exécution de la tâche τ_i, ce qui représente une augmentation de la fréquence d'horloge du coeur. Les deux dernières contraintes permettent de s'assurer que la modification du WCET est compatible avec une fréquence d'horloge correspondante à l'un des points de fonctionnement du cœur C qui est dans l'ensemble $[F_1 \dots F_{max}]$ (Section 3.2.3). Cette formulation permet de minimiser la modification de temps d'exécution et exploite la flexibilité possible des périodes.

3.3.3 Problème de contraintes énergétiques

Puisque $RTSys$ utilise une batterie comme source d'alimentation, il faut que son fonctionnement s'adapte à la quantité d'énergie disponible et au temps supposé restant jusqu'à la prochaine recharge. Après l'application de $RE\text{-}CONF$, l'utilisation du processeur sera probablement augmentée. Vu que la consommation d'énergie est proportionnelle à l'utilisation du processeur [2, 15], $RTSys$ consomme plus d'énergie et il peut être incapable de s'exécuter jusqu'à la prochaine recharge et par conséquent il pourrait violer sa

contrainte énergétique. Dans ce contexte, nous proposons de réduire l'utilisation du processeur en allongeant les périodes des tâches afin de minimiser la consommation d'énergie. Nous formalisons cette idée par l'ajout de l'Eq. 3.6 au système $Pb1$.

$$
Pb2 = \begin{cases}
\text{Minimiser } \sum_{i=1}^{M} \Delta W_i \\
t.q. \\
\quad \sum_{i=1}^{N} \frac{W_i - \Delta W_i}{T_i + \Delta T_i} \leq 1 \\
\quad W_i - \Delta W_i > 0, \ \forall \ i \in [1 \dots N] \\
\quad T_i + \Delta T_i \leq T_{i_{max}}, \ \forall \ i \in [1 \dots N] \\
\quad F_n * W_i/(W_i - \Delta W_i) \geq F_1 \\
\quad F_n * W_i/(W_i - \Delta W_i) \leq F_{max} \\
\quad \left(\sum_{i=1}^{N} \frac{W_i - \Delta W_i}{T_i + \Delta T_i} \right)^2 \leq P_{Limit} \qquad (Eq.3.6)
\end{cases}
$$

$Pb1$ et $Pb2$ peuvent être résolus par un solveur CPLEX [81] qui peut fournir une solution optimale si elle existe. Toutefois, il n'est pas raisonnable de mettre en œuvre ce genre de résolution dans un système embarqué puisqu'il prend un temps énorme pour produire des solutions à ce genre de systèmes. Comme nous avons précisé dans la Section 2.3.3.3, l'ordonnancement se fait en ligne, il n'est pas possible d'ordonnancer les tâches en utilisant un solveur embarqué. Par conséquent, nous présentons dans la section suivante des heuristiques efficaces pour un système temps-réel embarqué.

3.4 Contribution et heuristiques d'ordonnancement proposées

Lorsque $RE\text{-}CONF$ est appliqué, le taux d'utilisation du cœur C peut augmenter et par conséquent les contraintes temps-réel et énergétique peuvent être violées [2, 40] et $RTSys$ devient non faisable. Pour réobtenir la faisabilité du système après l'événement $RE\text{-}CONF$, nous suggérons d'adapter les paramètres des tâches afin de diminuer le taux d'utilisation du cœur. Cela se faire par l'allongement de périodes des tâches T_i ou bien par la diminution des WCETs W_i. Pour modifier ces paramètres, nous proposons cinq heuristiques (A, B, C, D, E) [93]. Chaque heuristique (sauf l'heuristique E) est basée sur deux étapes importantes :

1. Regroupement des tâches dans des $Packs$: Nous regroupons les tâches dans des $Packs$ de sorte qu'elles soient "similaires" au niveau de leur période (ou leur WCET). La façon de construire les $Packs$ est présentée et détaillée dans la section suivante en résolvant $Pb3$ pour faire le regroupement des tâches selon leur période (section 3.4.1.1) et $Pb4$ pour le regroupement de tâches selon leur WCET (section 3.4.1.2).

2. Affectation des périodes (ou WCETs) : Affectation d'une valeur unique pour les périodes (ou WCETs) du premier $Pack$. Les autres $Packs$ ont des périodes (ou WCETs) multiples. Exemple : Si $Pack_1$ contient des tâches dont les périodes sont égales à 10, alors les périodes de tâches dans $Pack_2$ sont égales à $2 * 10 = 20$, ..., les périodes de tâches dans $Pack_x$ sont égales à $x * 10$ (x est un entier positif), etc. Cette partie est également expliquée dans la section suivante.

Nous montrons dans la suite comment les tâches sont regroupées dans des $Packs$. En effet, l'adaptation des périodes et/ou de temps d'exécution des tâches est nécessaire pour faire redescendre le taux d'utilisation du cœur afin de satisfaire la ou les contrainte(s) violée(s). Par ailleurs, chaque modification d'un paramètre de tâche τ_i a un coût noté $CoûtModif_{\tau_i}$ qui est égal à la différence entre la nouvelle valeur et la valeur initiale du paramètre modifié. Ainsi, le coût total de modification $CoûtTotalModif$ est la somme de tous les $CoûtModif_{\tau_i}$.
La description des heuristiques est comme suit :

- Heuristique A : en regroupant les tâches dans des packs, nous allongeons les périodes de tâches pour tenter de satisfaire la contraintes temps-réel,
- Heuristique B : en regroupant les tâches dans des packs, nous diminuons les WCETs de tâches pour tenter de satisfaire la contrainte temps-réel,
- Heuristique C : en regroupant les tâches dans des packs, nous allongeons les périodes de tâches pour tenter de satisfaire la contrainte énergétique,
- Heuristique D : en regroupant les tâches dans des packs, nous diminuons les WCETs de tâches pour tenter de satisfaire la contrainte énergétique, et
- Heuristique E : suppression de tâches selon leur facteur d'importance I_i.

Cette section présente une stratégie d'ordonnancement basée sur ces heuristiques afin de réobtenir la faisabilité du système après un scénario de reconfiguration $RE\text{-}CONF$. La Figure 3.1 présente un aperçu global de la stratégie que nous avons développée.

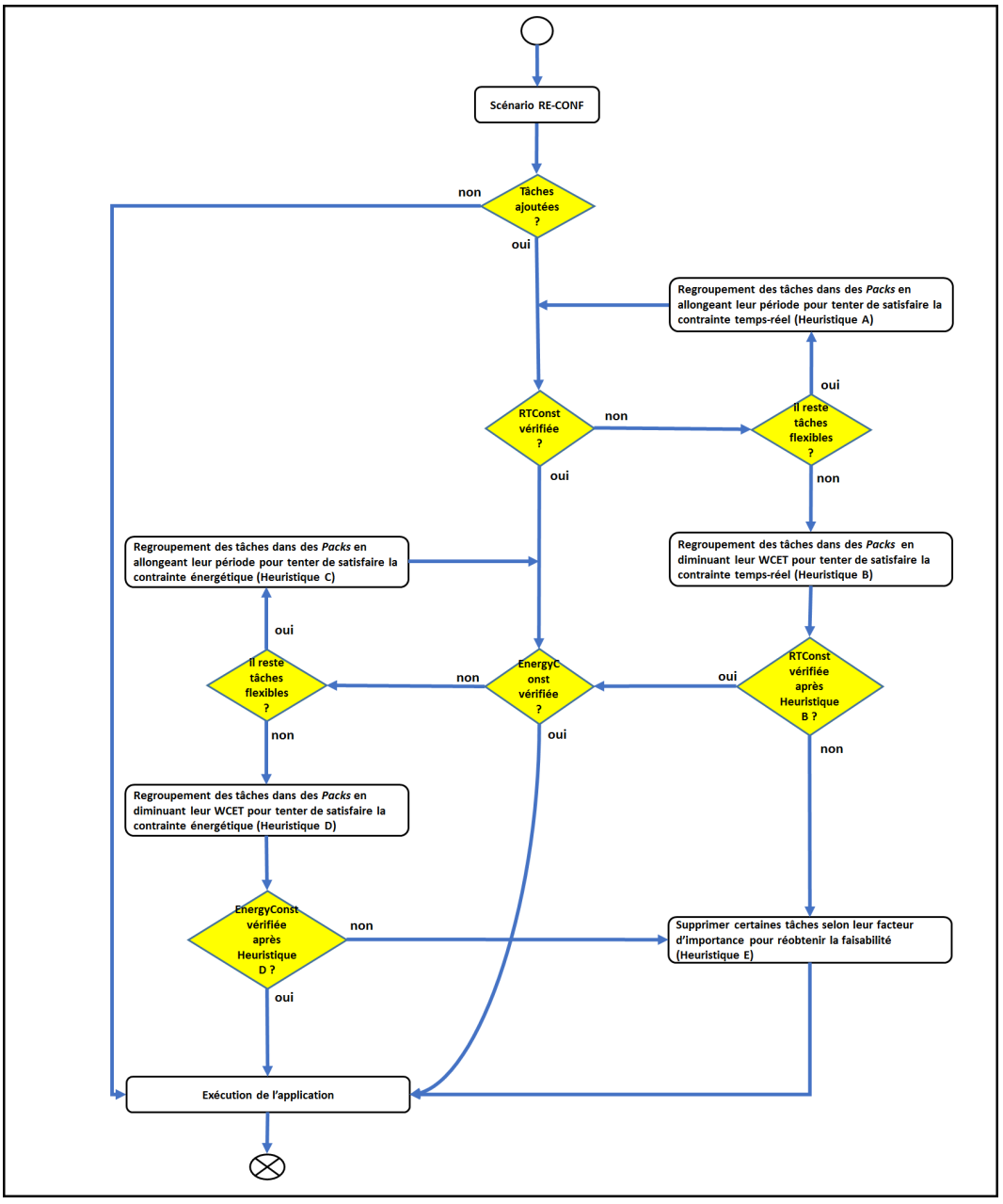

FIGURE 3.1: Aperçu global sur la stratégie d'ordonnancement Mono-cœur.

3.4.1 Résolution du problème de contraintes temps-réel

Après un ou plusieurs *RE-CONF*, la charge du processeur augmente et peut dépasser 1. Dans ce cas, *RTSys* ne respecte pas sa contrainte temps-réel et il n'est plus ordonnançable. Nous détaillons dans cette section les solutions pour modifier les paramètres des anciennes et récentes tâches après *RE-CONF* afin de résoudre le problème de violation de contrainte temps-réel.

3.4.1.1 Heuristique A : Modification des périodes des tâches sous contraintes temps-réel

Si le cœur C n'est pas faisable après $RE\text{-}CONF$, alors nous proposons d'allonger les périodes des tâches qui s'exécutent sur C. Afin de satisfaire la contrainte temps-réel, nous regroupons tout d'abord toutes les tâches dans des $Packs$ selon leur période. Cette idée est basée sur deux étapes :

Étape 1 : Construction des packs

Pour construire les packs, il est nécessaire de chercher la valeur de période convenable pour le premier $Pack\ Pk_1^T$ qui minimise le coût de modification des paramètres pour l'ensemble du système. Nous formalisons ce problème par :

$$Pb3 \begin{cases} \text{Minimiser} \sum_{i=1}^{N} \left((Pk_1^T - (T_i \ mod \ Pk_1^T)) \ mod \ Pk_1^T \right) \\ t.q. \\ \qquad Pk_1^T \geq Min(T_i), \ \forall \ i \in [1 \dots N] \end{cases}$$

La fonction objectif du $Pb3$ permet de trouver Pk_1^T qui engendre un coût total de modification minimal. Pour calculer le coût de modification d'une telle période T_i pour un pack Pk_1^T donné, il suffit d'appliquer cette formule : $\left((Pk_1^T - (T_i \ mod \ Pk_1^T)) \ mod \ Pk_1^T \right)$.

Exemple : disons qu'on va commencer par des packs multiples de 3, cela veut dire que Pk_1^T=3.

- Pour une tâche τ_i qui a une période T_i = 20, le coût de modification est égal à [(3 -(20 mod 3)) mod 3] = (3-2) mod 3 = 1. Pourquoi ? Puisque τ_i ne peut être que dans le septième pack (pack de 21), donc le coût est égal à 21-20=1.
- Pour une tâche τ_i qui a une période T_i = 3, le coût de modification est égal à [(3 -(3 mod 3)) mod 3] = (3-0) mod 3 = 0. Pourquoi ? Puisque τ_i ne peut être que dans le premier pack (pack de 3), donc le coût est égal à 3-3=0.

Une fois que le pack Pk_1^T est calculé, nous construisons les autres $Packs$ comme suit : $Pack_x^T, x \geq 1$, contient les tâches qui ont des périodes dans $[(x-1) * Pk_1^T + 1 \quad ; \quad x * Pk_1^T]$, $x \in \mathbb{N}^+$. La première valeur de x est 1. Toutes les périodes des tâches groupées dans $Pack_x^T$ auront une seule période qui égale à $x * Pk_1^T$. Dès que toutes les tâches sont affectées aux

différents $Pack$, x ne sera plus incrémentée.
Exemple : $Pack_1^T$ contient les tâches qui ont des périodes dans $[1 \quad ; \quad Pk_1^T]$, $Pack_2^T$ contient les tâches qui ont des périodes dans $[Pk_1^T + 1 \quad ; \quad 2 * Pk_1^T]$, ..., $Pack_x^T$ contient les tâches qui ont des périodes dans $[(x-1) * Pk_1^T + 1 \quad ; \quad x * Pk_1^T]$, etc.

Par ailleurs, chaque nouvelle période ne doit pas dépasser sa période maximale. Cette contrainte est décrite comme suit : $\forall \tau_i \mid \tau_i \in Pack_x^T, T_{i_{max}} \geq Pk_x^T$. Nous notons que si cette contrainte n'est pas respectée pour l'une des tâches, alors on a pas de solution et nous passons à l'exécution de l'heuristique B (Figure 3.1).

Après avoir construit les packs, on vérifie si les nouvelles valeurs satisfont la contrainte temps-réel ou pas. Si cette contrainte reste encore violée, on passe à la deuxième étape qui est la recherche des nouvelles valeurs des périodes des tâches tout en se basant sur les packs.

Étape 2 : Recherches des nouvelles périodes des tâches

Afin de satisfaire la contrainte temps-réel en utilisant l'algorithme d'ordonnancement EDF, il est nécessaire que $\sum_{i=1}^{N} \frac{W_i}{T_i} \leq 1$. Et comme les périodes de tâches sont multiples de celles du premier $Pack$, on a :

$$\sum_{\tau_i \in Pack_1^T} \frac{W_i}{Pk_1^T} + \ldots + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x.Pk_1^T} \leq 1$$

Ainsi,

$$\frac{1}{Pk_1^T} * \left(\sum_{\tau_i \in Pack_1^T} W_i + \ldots + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x} \right) \leq 1$$

Alors,

$$Pk_1^T \geq \sum_{\tau_i \in Pack_1^T} W_i + \ldots + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x}$$

Comme les périodes sont des entiers, nous obtiendrons donc

$$Pk_1^T = \left\lceil \sum_{\tau_i \in Pack_1^T} W_i + \ldots + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x} \right\rceil \tag{3.7}$$

Avec Pk_1^T la nouvelle période affectée à toutes les tâches de $Pack_1^T$. $x * Pk_1^T$ est la nouvelle période affectée à toutes les tâches de x^{ime} $Pack_x^T$.

Après l'allongement des périodes de tâches, la contrainte temps-réel peut être satisfaite. Si elle ne l'est pas, il faut passer à la seconde heuristique de la stratégie qui consiste à réduire les temps d'exécution en augmentant, dans la mesure du possible, la fréquence d'horloge du processeur.

Étape 1 : En résolvant le problème $Pb3$ par un solveur CPLEX, nous obtenons une et une seule valeur de Pk_1^T. La fonction objectif du $Pb3$ permet de trouver la valeur de Pk_1^T qui minimise le coût total de la modification. Nous appliquons donc l'Étape 1 à l'étude de cas *RE-CONF* (Section 3.3.1) afin trouver Pk_1^T. Selon cette étude de cas, 15 et 40 représentent respectivement les valeurs min et max des périodes de l'ensemble Π. Ainsi, nous calculons pour chaque pack combien il engendre de coût total de modification. Selon la Figure 3.2, Pk_1^T = 15 est le meilleur résultat en terme de coût de modification.
Par ailleurs, puisque la plus grande période est égale à 40, on doit donc construire trois packs pour grouper toutes les tâches tel que :

- $Pack_1^T$ contient les tâches qui ont des périodes dans [1 ; 15],
- $Pack_2^T$ contient les tâches qui ont des périodes dans [16 ; 30], et
- $Pack_3^T$ contient les tâches qui ont des périodes dans [31 ; 45].

Autrement dit,

- $Pack_1^T$ groupe la tâche τ_2,
- $Pack_2^T$ groupe les tâches τ_3, τ_5 et τ_6, et
- $Pack_3^T$ groupe les tâches τ_1 et τ_4

Les nouvelles périodes des tâches sont : T_1 = 45, T_2 = 15, T_3 = 30, T_4 = 45, T_5 = 30 et T_6 = 30. Nous pouvons alors à nouveau vérifier les conditions d'ordonnancabilité du système : $U_C = \sum_{i=1}^{6} \frac{W_i}{T_i} = \frac{4}{45} + \frac{6}{15} + \frac{3}{30} + \frac{4}{45} + \frac{5}{30} + \frac{6}{30} = 1.04 > 1$. L'utilisation du processeur a été abaissée, mais le système reste encore non ordonnançable. Dans ce cas, nous passons à l'Étape 2 pour trouver une nouvelle valeur de Pk_1^T.

Pk^T_1	*CoûtTotalModif*	Pk^T_1	*CoûtTotalModif*
15	26	28	83
16	39	29	63
17	52	30	71
18	65	31	79
19	78	32	87
20	71	33	95
21	83	34	103
22	95	35	111
23	107	36	119
24	119	37	127
25	56	38	135
26	65	39	143
27	74	40	71

FIGURE 3.2: Coût de modification en résolvant $Pb3$.

Étape 2 : Nous appliquons l'Étape 2 de l'heuristique A à l'étude de cas *RECONF* (Section 3.3.1) pour trouver la nouvelle valeur de Pk_1^T afin de satisfaire la contrainte temps-réel. Puisque les *Packs* sont bien construits dans l'Étape 1, nous commençons par appliquer l'Eq. 3.7, on a : $Pk_1^T = \left\lceil \sum_{\tau_i \in Pack_1^T} W_i + \ldots + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x} \right\rceil = \left\lceil \sum_{\tau_i \in Pack_1^T} W_i + \sum_{\tau_i \in Pack_2^T} \frac{W_i}{2} + \sum_{\tau_i \in Pack_3^T} \frac{W_i}{3} \right\rceil = \left\lceil 6 + \frac{3}{2} + \frac{5}{2} + \frac{6}{2} + \frac{4}{3} + \frac{4}{3} \right\rceil = \lceil 15.666 \rceil = 16.$ D'où,

- $Pack_1^T$ contient les tâches qui ont des périodes dans [1 ; 16],
- $Pack_2^T$ contient les tâches qui ont des périodes dans [17 ; 32], et
- $Pack_3^T$ contient les tâches qui ont des périodes dans [33 ; 48].

Autrement dit,

- $Pack_1^T$ groupe la tâche τ_2,
- $Pack_2^T$ groupe les tâches τ_3, τ_5 et τ_6, et
- $Pack_3^T$ groupe les tâches τ_1 et τ_4

Les nouvelles périodes des tâches deviennent donc : $T_1 = 48$, $T_2 = 16$, $T_3 = 32$, $T_4 = 48$, $T_5 = 32$ et $T_6 = 32$. Ainsi, nous vérifions l'ordonnançabilité du système : $U_C = \sum_{i=1}^{6} \frac{W_i}{T_i} = \frac{4}{48} + \frac{6}{16} + \frac{3}{32} + \frac{4}{48} + \frac{5}{32} + \frac{6}{32} = 0.979 \leq 1$. Le système devient ordonnançable avec un coût total de modification est égal à 39 ($Pk_1^T = 16$).

3.4.1.2 Heuristique B : Modification des WCETs des tâches sous contraintes temps-réel

En traitant le même problème qui est la violation de contraintes temps-réel du cœur C, nous proposons de résoudre ce problème différemment en réduisant les WCETs des tâches afin de baisser le taux d'utilisation.

Cette heuristique engendre une augmentation de la fréquence d'horloge du coeur, c'est pour cela elle ne peut être appliquée, par la stratégie, qu'après l'heuristique A qui ne provoque pas l'architecture matérielle du système.

Pour baisser donc le taux d'utilisation du cœur C, nous proposons d'utiliser la même technique que celle présentée dans l'heuristique A, mais cette fois-ci le regroupement de tâches est selon leur WCET.

Étape 1 : Construction des packs

Tout d'abord, nous essayons de trouver la valeur de Pk_1^W pour regrouper les tâches selon leurs WCETs. Soit la formalisation suivante :

$$Pb4 \begin{cases} \text{Minimiser } \sum_{i=1}^{N} \left((Pk_1^W - (W_i \ mod \ Pk_1^W)) \ mod \ Pk_1^W \right) \\ t.q. \\ \qquad Pk_1^W \geq Min(W_i), \ \forall \ \text{i} \in [1..N] \end{cases}$$

La fonction objectif du $Pb4$ permet de chercher Pk_1^W qui minimise le coût total de modification lors de construction des packs.

Une fois que Pk_1^W est trouvé, nous pouvons donc construire les paquets de tâches comme suit : $Pack_x^W$ contient les tâches qui ont des WCETs dans $[(x-1) * Pk_1^W + 1 \quad ; \quad x * Pk_1^W]$. Cette première étape peut engendrer une augmentation des WCETs. Pour cela, nous passons directement à l'Étape 2, sans vérifier l'ordonnançabilité du système, pour diminuer les WCETs.

Étape 2 : Recherches des nouvelles WCETs des tâches

Pour satisfaire la contrainte temps-réel, il faut que $\sum_{i=1}^{N} \frac{W_i}{T_i} \leq 1$. Puisque les WCETs W_i sont les multiples de la valeur Pk_1^W, on a :

$$\sum_{\tau_i \in Pack_1^W} \frac{Pk_1^W}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x.Pk_1^W}{T_i} \leq 1$$

Ainsi,

$$Pk_1^W * \left(\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x}{T_i} \right) \leq 1$$

Donc,

$$Pk_1^W \leq \frac{1}{\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x}{T_i}}$$

Comme les WCETs sont des entiers, nous obtiendrons donc

$$Pk_1^W = \left\lfloor \frac{1}{\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x}{T_i}} \right\rfloor \tag{3.8}$$

Pk_1^W est la nouvelle valeur de WCET affectée à toutes les tâches du premier paquet $Pack_1^W$. En conséquence, $x * Pk_1^W$ sera la nouvelle valeur de WCET affectée à toutes les tâches du x^{ime} $Pack_x^W$.

Par ailleurs, il est nécessaire de s'assurer que la modification du WCET est compatible avec une fréquence d'horloge correspondante à l'un des points de fonctionnement du cœur. Cette contrainte est décrite comme suit : $\forall \tau_i \mid \tau_i \in Pack_x^W, F_1 \leq (\frac{Pk_x^W}{W_i} * F_n) \leq F_{max}$. Si cette contrainte n'est pas vérifiée, alors on a pas de solution et nous passons à l'exécution de l'heuristique E (Figure 3.1).

Une fois que le temps d'exécution des tâches est réduit, l'utilisation du processeur sera diminuée et par conséquent la contrainte temps-réel peut être satisfaite. Si elle ne l'est pas, alors il faut passer à la troisième étape de la stratégie qui consiste à supprimer des tâches en fonction de leur importance.

Étape 1 : La fonction objectif du $Pb4$ permet de trouver la valeur de Pk_1^W qui minimise le coût total de modification. Nous appliquons $Pb4$ à l'étude de cas $RE\text{-}CONF$ (Section 3.3.1) afin d'expliquer comment trouver l'unique valeur de Pk_1^W. Selon cette étude de cas, 3 et 6 représentent respectivement les valeurs min et max des WCETs. Ainsi, nous calculons pour chaque pack combien il engendre de coût total de modification. Selon la Figure 3.3, Pk_1^W = 3 est le meilleur résultat en terme de coût de modification. On a donc deux packs :

- $Pack_1^W$ contient les tâches qui ont des WCETs dans [1 ; 3], et
- $Pack_2^W$ contient les tâches qui ont des WCETs dans [4 ; 6].

Autrement dit,

- $Pack_1^W$ groupe la tâche τ_3,
- $Pack_2^W$ groupe les tâches τ_1, τ_2, τ_4, τ_5 et τ_6

Les nouvelles valeurs de WCETs des tâches sont : W_1 = 6, W_2 = 6, W_3 = 3, W_4 = 6, W_5 = 6 et W_6 = 6. Cette étape permet de construire les packs pour pouvoir appliquer l'Eq. 3.8. Maintenant, pour diminuer les WCETs, on passe à l'Étape 2.

Pk^W_1	*CoûtTotalModif*
3	5
4	8
5	12
6	8

FIGURE 3.3: Coût de modification en résolvant $Pb4$.

Étape 2 : Nous appliquons l'Étape 2 de l'heuristique B à l'étude de cas *RE-CONF* (Section 3.3.1) pour trouver la nouvelle valeur de Pk_1^W afin de satisfaire la contrainte temps-réel. Puisque les *Pack* sont bien construits, nous commençons par appliquer l'Eq. 3.8, on a : $Pk_1^W = \left\lfloor \frac{1}{\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x}{T_i}} \right\rfloor = \left\lfloor \frac{1}{\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \sum_{\tau_i \in Pack_2^W} \frac{2}{T_i}} \right\rfloor = \left\lfloor \frac{1}{\frac{1}{29} + \frac{2}{40} + \frac{2}{40} + \frac{2}{20} + \frac{2}{25} + \frac{2}{15}} \right\rfloor = \lfloor 2.24 \rfloor = 2.$
D'où, les WCETs des tâches du $Pack_1^W$ auront la valeur 2 et les WCETs des tâches du $Pack_2^W$ auront 4.
Les nouvelles WCETs des tâches sont : $W_1 = 4$, $W_2 = 4$, $W_3 = 2$, $W_4 = 4$, $W_5 = 4$ et $W_6 = 4$.
Cette modification au niveaux des WCETs engendre une augmentation de la fréquence d'horloge (nominale) du cœur C lors de l'exécution de certaines tâches. Similairement à l'article [94], nous supposons que la fréquence nominale $F_n = 2Ghz$ avec un pas de fréquence de $100Mhz$, avec :

- 2Ghz est la fréquence minimale du cœur C qui est notée par $F1$ dans la formalisation ILP (Section 3.3.2).
- 3.2Ghz est la fréquence maximale du cœur C qui est notée par F_{max} dans la formalisation ILP (Section 3.3.2).

Autrement dit :

- Pour exécuter τ_1, nous gardons la fréquence nominale $F_n = 2Ghz$ (pas de changement au niveau de W_1),
- Pour exécuter τ_2, nous augmentons la fréquence à $\frac{6}{4} * F_n = 3Ghz$,
- Pour exécuter τ_3, nous augmentons la fréquence à $\frac{6}{4} * F_n = 3Ghz$,
- Pour exécuter τ_4, nous gardons la fréquence nominale $F_n = 2Ghz$,
- Pour exécuter τ_5, nous augmentons la fréquence à $\frac{5}{4} * F_n = 2.5Ghz$, et
- Pour exécuter τ_6, nous augmentons la fréquence à $\frac{6}{5} * F_n = 3Ghz$.

Étant donné que les nouvelles fréquences correspondent aux points de fonctionnement du cœur C, nous pouvons donc vérifier l'ordonnançabilité du système : $U_C = \sum_{i=1}^{6} \frac{W_i}{T_i} = \frac{4}{40} + \frac{4}{15} + \frac{2}{29} + \frac{4}{40} + \frac{4}{20} + \frac{4}{25} = 0.895 \leq 1$. Le système redevient donc ordonnançable.
Dans le cas où l'une des nouvelles fréquences ne correspond pas à l'un des points de fonctionnement du cœur, alors cette heuristique ne résout pas le problème et il faut passer à l'heuristique suivante qui est la suppression de certaines tâches.

3.4.2 Résolution du problème de contraintes énergétique

Selon [2, 7, 95], la charge du processeur doit être diminuée afin de garantir une consommation d'énergie acceptable pour que $RTSys$ puisse s'exécuter. Nous suggérons également la modification des périodes ou des WCETs afin que $RTSys$ puisse poursuivre son exécution sans violation de sa contrainte énergétique après un ou plusieurs $RE\text{-}CONF$. En effet, nous utilisons la même technique de regroupement de tâches dans des $Packs$ afin de proposer deux autres heuristiques qui résolvent la contrainte énergétique. Une heuristique basée sur le regroupement des tâches selon leur période et l'autre selon leur WCET.

Ainsi, pour satisfaire la contrainte énergétique, il est nécessaire de s'assurer que la puissance consommée à l'instant t soit inférieure ou égale à la puissance limite $P_{Limit}(t)$ (contrainte formulée dans la Section 3.2.3). Autrement dit, il faut que $k * U^2 \leq P_{Limit}(t)$

D'où,

$$U \leq \sqrt{\frac{P_{Limit}(t)}{k}}$$

Donc,

$$\sum_{i=1}^{N} \frac{W_i}{T_i} \leq \sqrt{\frac{P_{Limit}(t)}{k}}$$

3.4.2.1 Heuristique C : Modification des périodes des tâches sous contraintes d'énergie d'un système mono-cœur

Si $RTSys$ applique un ou plusieurs $RE\text{-}CONF$, il risque de violer sa contrainte énergétique vu que la consommation va augmenter. Une troisième solution est donc proposée en utilisant la même technique que celle présentée dans l'heuristique A (notion de $Packs$ pour les tâches en fonction des périodes). Nous résolvons donc le même système $Pb3$ (Section 3.4.1.1) pour chercher la première valeur Pk_1^T afin de construire tous les $Pack$. Une fois que Pk_1^T est calculé, nous construisons les autres $Packs$, et nous aurons donc :

$$\sum_{\tau_i \in Pack_1^T} \frac{W_i}{Pk_1^T} + \ldots + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x.Pk_1^T} \leq \sqrt{\frac{P_{Limit}(t)}{k}}$$

Ainsi,

$$\frac{1}{Pk_1^T} * \left(\sum_{\tau_i \in Pack_1^T} W_i + \ldots + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x} \right) \leq \sqrt{\frac{P_{Limit}(t)}{k}}$$

Alors,

$$Pk_1^T \geq \frac{\sum_{\tau_i \in Pack_1^T} W_i + ... + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x}}{\sqrt{\frac{P_{Limit}(t)}{k}}}$$

Puisque les périodes sont des entiers, nous obtiendrons donc :

$$Pk_1^T = \left\lceil \frac{\sum_{\tau_i \in Pack_1^T} W_i + ... + \sum_{\tau_i \in Pack_x^T} \frac{W_i}{x}}{\sqrt{\frac{P_{Limit}(t)}{k}}} \right\rceil \tag{3.9}$$

Pk_1^T est la nouvelle période affectée à toutes les tâches de $Pack_1^T$.
$x * Pk_1^T$ est la nouvelle période affectée à toutes les tâches de x^{ime} $Pack_x^T$.

Par ailleurs, chaque nouvelle période ne doit pas dépasser sa période maximale. Cette contrainte est décrite comme suit : $\forall \tau_i \mid \tau_i \in Pack_x^T, T_{i_{max}} \geq Pk_x^T$.

Après l'allongement des périodes des tâches, la contrainte énergétique peut être satisfaite. Si elle ne l'est pas, alors nous suggérons une deuxième solution qui consiste à réduire les WCETs des tâches sous contrainte d'énergie.

3.4.2.2 Heuristique D : Modification des WCETs des tâches sous contraintes d'énergie d'un système mono-cœur

Cette heuristique est la deuxième alternative pour rationner l'énergie restante en jouant sur les WCETs des tâches pour réduire le taux d'utilisation du cœur afin de baisser la consommation d'énergie. Pour ce faire, nous cherchons la première valeur de WCET du premier *Packs* Pk_1^W par la résolution du $Pb4$ (Section 3.4.1.2) afin de construire tous les *Pack*. Après avoir trouvé la valeur de Pk_1^W, nous pouvons donc construire les *Packs* de tâches, et nous pouvons donc dire que :

$$\sum_{\tau_i \in Pack_1^W} \frac{W_i}{Pk_1^W} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{W_i}{x.Pk_1^W} \leq \sqrt{\frac{P_{Limit}(t)}{k}}$$

Ainsi,

$$Pk_1^W * \left(\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x}{T_i} \right) \leq \sqrt{\frac{P_{Limit}(t)}{k}}$$

Alors,

$$Pk_1^W \leq \frac{\sqrt{\frac{P_{Limit}(t)}{k}}}{\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x}{T_i}}$$

Comme les WCETs sont des entiers, nous obtiendrons :

$$Pk_1^W = \left\lfloor \frac{\sqrt{\frac{P_{Limit}(t)}{k}}}{\sum_{\tau_i \in Pack_1^W} \frac{1}{T_i} + \ldots + \sum_{\tau_i \in Pack_x^W} \frac{x}{T_i}} \right\rfloor \tag{3.10}$$

Une fois Pk_1^W calculé, nous vérifions que la modification du WCET est compatible avec une fréquence d'horloge correspondante à l'un des points de fonctionnement du cœur :

$\forall \tau_i \mid \tau_i \in Pack_x^W, F_1 \leq (\frac{Pk_x^W}{W_i} * F_n) \leq F_{max}$

Une fois que le temps d'exécution des tâches est réduit, l'utilisation du processeur sera diminuée et par conséquent la contrainte temps-réel peut être satisfaite. Si elle ne l'est pas, alors il faut passer à la troisième étape de la stratégie qui consiste à supprimer des tâches en fonction de leur importance.

3.4.2.3 Heuristique E : Suppression de tâches

Après avoir modifié les paramètres de tâches en appliquant les heuristiques proposées (i.e. après l'exploitation de la flexibilité des tâches et la diminution de leur WCET), *RTSys* peut être encore non faisable, nous proposons alors de supprimer des tâches du système, ce qui constitue la seule possibilité de rendre le système ordinnançable. Le choix des tâches qui doivent être supprimées se fait selon leur facteur d'importance I_i. Dans le cas où il existe plusieurs tâches qui possèdent le même facteur d'importance, la tâche τ_i possédant un taux d'utilisation u_{τ_i} le plus élevé, devient prioritaire pour être supprimée. Avec cette proposition, c'est la dernière chance que le système puisse redevenir faisable.

3.4.3 Proposition d'une stratégie d'ordonnancement de tâches

Un *RTSys* est dit faisable s'il satisfait les contraintes **RTConst** et **EnergyConst**. Pour assurer sa faisabilité après l'événement *RE-CONF*, nous proposons une stratégie basée sur la notion de classification de tâches. Cette stratégie implémente les cinq heuristiques (A, B, C, D et E) et elle évalue dans un premier temps l'allongement des périodes des tâches pour tenter de satisfaire la contrainte temps-réel, et cela tant que ces périodes n'atteignent pas leur période maximale (heuristique A). Si *RTSys* reste encore non faisable après l'application de l'heuristique A, alors la stratégie propose de réduire le temps

d'exécution de certaines tâches (heuristique B). Finalement, si le système n'est toujours pas faisable, la stratégie propose de supprimer certaines tâches jugées non importantes (heuristique E). Une fois la contrainte temps-réel vérifiée, nous passons à l'évaluation de la contrainte énergétique en utilisant le même principe : Si cette contrainte n'est pas respectée, nous allongeons les périodes de tâches en appliquant l'heuristique C. Si elle est encore violée, alors nous proposons d'appliquer l'heuristique D qui permet de réduire le temps d'exécution de certaines tâches. Au pire des cas ($RTSys$ reste toujours non faisable), l'heuristique E doit être appliquée pour supprimer certaines tâches afin de reobtenir la faisabilité.

Par ailleurs, il est aussi important de réobtenir la faisabilité du système avec un coût de modification minimum. L'Algorithme 1 montre comment les heuristiques s'appliquent dans chaque cas de violation de contrainte. Dans le cas où les deux contraintes sont en même temps violées, notre algorithme propose dans un premier temps de tester les heuristiques A (pour la contrainte temps-réel) et C (pour la contrainte énergétique) et d'appliquer celle qui allonge plus les périodes des tâches puisqu'elle peut satisfaire simultanément les deux contraintes. Si $RTSys$ reste encore non faisable, dans ce cas il choisit une heuristique entre B et D et applique celle qui augmente plus la fréquence du processeur pour tenter de satisfaire les deux contraintes en même temps. D'un point de vue complexité, il a une complexité $O(N^2)$ puisque chaque heuristique (sauf E) est codée en utilisant deux boucles imbriquées pour parcourir les tâches et construire leurs packs [93]. Avant de décrire l'algorithme, nous utilisons les fonctions suivantes :

- **Faisable(RTSys)** : Retourne Vrai si $RTSys$ est faisable.
- **Max{Heuristique X, Heuristique Y}** : Sélectionne l'une des deux heuristiques X et Y possédant la plus grande Pk_1^T.
- **Min{Heuristique X, Heuristique Y}** : Sélectionne l'une des deux heuristiques X et Y possédant la plus petite Pk_1^W.
- **Appliquer(Heuristique X)** : Exécute l'heuristique X

Afin de montrer l'efficacité de la stratégie développée, nous proposons dans la section suivante un simulateur qui implémente les heuristiques proposées.

Algorithm 1 Stratégie d'ordonnancement appliquée après une ou des reconfigurations

```
1: Input : RTSys après l'événement RE − CONF
2: if (!RTConst) and (EnergyConst) then
3:     Appliquer (Heuristique A)
4:     if (!Faisable(RTSys)) then
5:         Appliquer (Heuristique B)
6:     end if
7: else if (RTConst) and (!EnergyConst) then
8:     Appliquer (Heuristique C)
9:     if (!Faisable(RTSys)) then
10:        Appliquer (Heuristique D)
11:    end if
12: else if (!RTConst) and (!EnergyConst) then
13:    RTEgPERIOD = Max{Heuristique A, Heuristique C}
14:    Appliquer (RTEgPERIOD)
15:    if (!Faisable(RTSys)) then
16:        if (!RTConst) and (!EnergyConst) then
17:            RTEgWCET = Min{Heuristique B, Heuristique D}
18:            Appliquer (RTEgWCET)
19:        else if (!RTConst) and (EnergyConst) then
20:            Appliquer (Heuristique B)
21:        else if (RTConst) and (!EnergyConst) then
22:            Appliquer (Heuristique D)
23:        end if
24:    end if
25: end if
26: if (!Faisable(RTSys)) then
27:    Appliquer (Heuristique E)
28: end if
29: Output : RTSys est faisable
```

3.5 Simulations et discussion

Cette section permet de d'évaluer, théoriquement, la performance de notre stratégie et de se situer par rapport aux articles [2, 15] et par rapport à la solution optimale obtenue par un solveur CPLEX [81]. Une génération aléatoire de systèmes ainsi que de tâches a été effectuée afin de pouvoir calculer le gain moyen offert par notre stratégie.

3.5.1 Étude comparative

Nous avons développé un simulateur $Reconf-Pack$ qui implémente nos solutions et celle de Wang et al. [2, 15]. Ce travail a été publié dans [96]. Le simulateur $Reconf-Pack$ (Figure 3.4) a pour objectif de comparer les solutions proposées dans ce chapitre et celles

proposées par Wang et al. [2, 15]. Toutes ces solutions sont basées sur la modification des paramètres en allongeant les périodes des tâches ou en réduisant leurs WCETs. Comme nous avons indiqué dans la section précédente, chaque modification d'un paramètre a un coût $CoûtModif_{\tau_i}$ qui est la différence entre la valeur initiale du paramètre et celle après sa modification. En calculant $CoûtModif_{\tau_i}$, $Reconf-Pack$ applique les deux approches et établit des courbes de comparaisons en terme de coût de modification.

Cette section permet de se situer par rapport à l'article [2] et par rapport à la solution optimale. En utilisant le simulateur $Reconf-Pack$, nous commençons d'évaluer l'heuristique A et de la comparer avec celle publiée par Wang et al. dans [2]. Nous proposons d'utiliser le même cas d'étude qui est représenté par les Figures 3.5 et 3.6 (Figures 7 et 8 de l'article [2]).

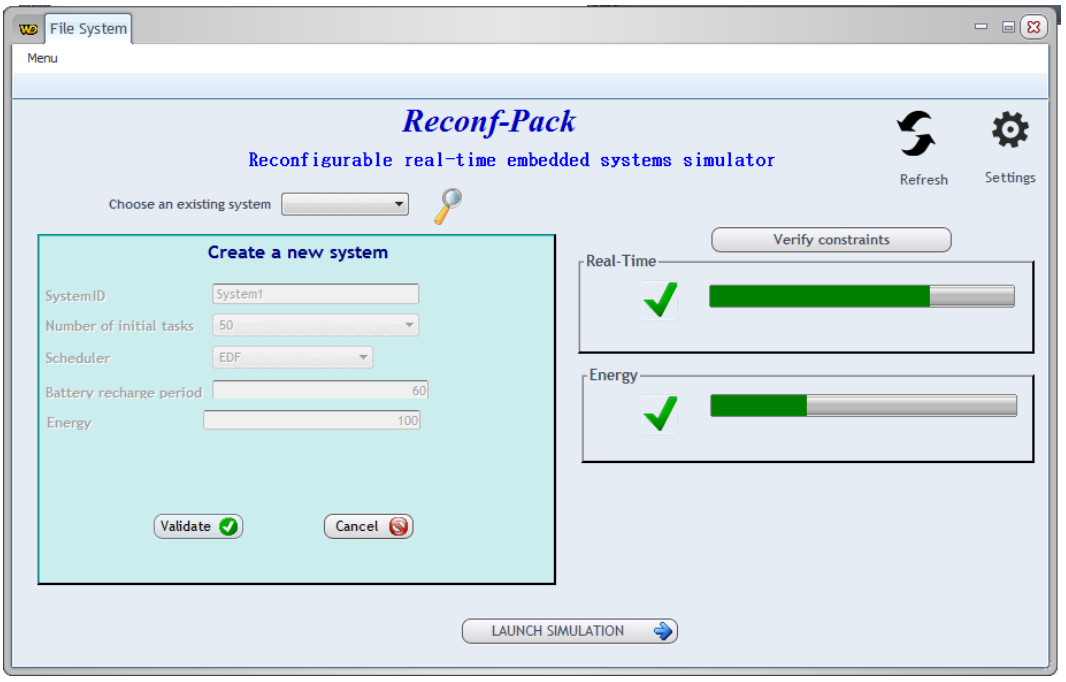

FIGURE 3.4: Interface graphique de l'outil de simulation $Reconf-Pack$.

Tâche	W_i	T_i	Tâche	W_i	T_i
τ_1	3	500	τ_{26}	3	900
τ_2	4	700	τ_{27}	4	920
τ_3	5	720	τ_{28}	5	940
τ_4	6	740	τ_{29}	2	960
τ_5	7	760	τ_{30}	3	980
τ_6	3	780	τ_{31}	4	980
τ_7	2	800	τ_{32}	5	800
τ_8	5	660	τ_{33}	6	640
τ_9	6	700	τ_{34}	4	700
τ_{10}	4	740	τ_{35}	8	680
τ_{11}	3	600	τ_{36}	4	700
τ_{12}	4	620	τ_{37}	5	720
τ_{13}	5	740	τ_{38}	6	740
τ_{14}	6	680	τ_{39}	4	760
τ_{15}	7	680	τ_{40}	8	780
τ_{16}	3	700	τ_{41}	4	800
τ_{17}	4	720	τ_{42}	5	820
τ_{18}	5	740	τ_{43}	6	840
τ_{19}	6	760	τ_{44}	7	860
τ_{20}	7	780	τ_{45}	8	880
τ_{21}	3	800	τ_{46}	4	900
τ_{22}	4	820	τ_{47}	5	920
τ_{23}	5	840	τ_{48}	3	940
τ_{24}	6	860	τ_{49}	3	960
τ_{25}	3	900	τ_{50}	4	980

FIGURE 3.5: Tâches initiales.

Tâche	W_i	T_i	Tâche	W_i	T_i
τ_{51}	3	205	τ_{66}	4	245
τ_{52}	4	215	τ_{67}	5	255
τ_{53}	5	225	τ_{68}	6	205
τ_{54}	6	235	τ_{69}	7	215
τ_{55}	7	245	τ_{70}	8	225
τ_{56}	3	215	τ_{71}	**4**	**255**
τ_{57}	4	225	τ_{72}	5	205
τ_{58}	5	235	τ_{73}	6	215
τ_{59}	6	245	τ_{74}	7	225
τ_{60}	7	255	τ_{75}	8	235
τ_{61}	3	225	τ_{76}	4	205
τ_{62}	4	235	τ_{77}	5	215
τ_{63}	5	245	τ_{78}	6	225
τ_{64}	6	255	τ_{79}	7	235
τ_{65}	7	205	τ_{80}	8	245

FIGURE 3.6: Tâches ajoutées.

La Figure 3.5 présente 50 tâches périodiques indépendantes exécutées par le système qui est initialement ordonnançable, sachant que le taux d'utilisation du cœur est égal à 0.71. Après un scénario de reconfiguration, qui ajoute 30 tâches (Figure 3.6), le taux d'utilisation augmente à 1.02 et le système devient non ordonnançable.

Afin de rendre le système ordonnançable après l'ajout des 30 tâches, Wang et al. [2] proposent soit de modifier les périodes T_i des tâches par l'attribution d'une période unique à toutes les tâches, soit de réduire les WCETs W_i par l'attribution d'une WCET unique pour toutes les tâches. Pour ce scénario, nous proposons d'appliquer l'heuristique A qui allonge les périodes de ces tâches et nous comparons avec la solution de Wang et al. [2]. Nous proposons d'appliquer cette heuristique dans les deux cas suivants :

- Cas 1 : Modification des périodes de tâches T_i sans prise en compte de leur période maximale $T_{i_{max}}$ (on se situe dans les mêmes conditions de l'article [2]).
- Cas 2 : Modification des périodes de tâches T_i avec prise en compte de leur période maximale $T_{i_{max}}$ (contrainte supplémentaire sur les périodes).

Cas 1 : Modification des périodes sans prise en compte de la contrainte $T_{i_{max}}$:

Après avoir appliqué la solution de Wang et al. [2], nous obtenons 401 comme période pour toutes les tâches avec U_C=0.983. Figure 3.7 illustre les nouveaux paramètres des tâches ainsi que le coût total de modification.

Après l'application de l'heuristique A et selon la Figure 3.8, nous obtenons Pk_1^T=255, il s'agit donc de quatre packs organisés comme suit :

- $Pack_1^T$ contient les tâches qui ont des périodes dans [1 ; 255],
- $Pack_2^T$ contient les tâches qui ont des périodes dans [256 ; 510],
- $Pack_3^T$ contient les tâches qui ont des périodes dans [511 ; 765], et
- $Pack_4^T$ contient les tâches qui ont des périodes dans [766 ; 1020].

Grâce à l'heuristique A, nous obtenons U_C=0.927 avec un coût total de modification est égal à 5965 (Figure 3.8) qui est très réduit par rapport à celui obtenu par la méthode présentée dans [2] qui est 24550 (Figure 3.7).

Tâche	W_i	Nouvelle T_i	*CoûtModif* τ_i	U_{ti}	Tâche	W_i	Nouvelle T_i	*CoûtModif* τ_i	U_{ti}
τ_1	3	401	99	0,007	τ_{41}	4	401	399	0,01
τ_2	4	401	299	0,01	τ_{42}	5	401	419	0,012
τ_3	5	401	319	0,012	τ_{43}	6	401	439	0,015
τ_4	6	401	339	0,015	τ_{44}	7	401	459	0,017
τ_5	7	401	359	0,017	τ_{45}	8	401	479	0,02
τ_6	3	401	379	0,007	τ_{46}	4	401	499	0,01
τ_7	2	401	399	0,005	τ_{47}	5	401	519	0,012
τ_8	5	401	259	0,012	τ_{48}	3	401	539	0,007
τ_9	6	401	299	0,015	τ_{49}	3	401	559	0,007
τ_{10}	4	401	339	0,01	τ_{50}	4	401	579	0,01
τ_{11}	3	401	199	0,007	τ_{51}	3	401	196	0,007
τ_{12}	4	401	219	0,01	τ_{52}	4	401	186	0,01
τ_{13}	5	401	339	0,012	τ_{53}	5	401	176	0,012
τ_{14}	6	401	279	0,015	τ_{54}	6	401	166	0,015
τ_{15}	7	401	279	0,017	τ_{55}	7	401	156	0,017
τ_{16}	3	401	299	0,007	τ_{56}	3	401	186	0,007
τ_{17}	4	401	319	0,01	τ_{57}	4	401	176	0,01
τ_{18}	5	401	339	0,012	τ_{58}	5	401	166	0,012
τ_{19}	6	401	359	0,015	τ_{59}	6	401	156	0,015
τ_{20}	7	401	379	0,017	τ_{60}	7	401	146	0,017
τ_{21}	3	401	399	0,007	τ_{61}	3	401	176	0,007
τ_{22}	4	401	419	0,01	τ_{62}	4	401	166	0,01
τ_{23}	5	401	439	0,012	τ_{63}	5	401	156	0,012
τ_{24}	6	401	459	0,015	τ_{64}	6	401	146	0,015
τ_{25}	3	401	499	0,007	τ_{65}	7	401	196	0,017
τ_{26}	3	401	499	0,007	τ_{66}	4	401	156	0,01
τ_{27}	4	401	519	0,01	τ_{67}	5	401	146	0,012
τ_{28}	5	401	539	0,012	τ_{68}	6	401	196	0,015
τ_{29}	2	401	559	0,005	τ_{69}	7	401	186	0,017
τ_{30}	3	401	579	0,007	τ_{70}	8	401	176	0,02
τ_{31}	4	401	579	0,01	τ_{71}	**4**	401	146	0,01
τ_{32}	5	401	399	0,012	τ_{72}	5	401	196	0,012
τ_{33}	6	401	239	0,015	τ_{73}	6	401	186	0,015
τ_{34}	4	401	299	0,01	τ_{74}	7	401	176	0,017
τ_{35}	8	401	279	0,02	τ_{75}	8	401	166	0,02
τ_{36}	4	401	299	0,01	τ_{76}	4	401	196	0,01
τ_{37}	5	401	319	0,012	τ_{77}	5	401	186	0,012
τ_{38}	6	401	339	0,015	τ_{78}	6	401	176	0,015
τ_{39}	4	401	359	0,01	τ_{79}	7	401	166	0,017
τ_{40}	8	401	379	0,02	τ_{80}	8	401	156	0,02
TOTAL								**24550**	**0,983**

FIGURE 3.7: Les nouvelles valeurs des périodes des tâches après l'application de la méthode présentée dans [2].

Tâche	W_i	Nouvelle T_i	*CoûtModif* τ_i	U_{ti}	Tâche	W_i	Nouvelle T_i	*CoûtModif* τ_i	U_{ti}
τ_1	3	510	10	0,006	τ_{41}	4	1020	220	0,004
τ_2	4	765	65	0,005	τ_{42}	5	1020	200	0,005
τ_3	5	765	45	0,007	τ_{43}	6	1020	180	0,006
τ_4	6	765	25	0,008	τ_{44}	7	1020	160	0,007
τ_5	7	765	5	0,009	τ_{45}	8	1020	140	0,008
τ_6	3	1020	240	0,003	τ_{46}	4	1020	120	0,004
τ_7	2	1020	220	0,002	τ_{47}	5	1020	100	0,005
τ_8	5	765	105	0,007	τ_{48}	3	1020	80	0,003
τ_9	6	765	65	0,008	τ_{49}	3	1020	60	0,003
τ_{10}	4	765	25	0,005	τ_{50}	4	1020	40	0,004
τ_{11}	3	765	165	0,004	τ_{51}	3	255	50	0,012
τ_{12}	4	765	145	0,005	τ_{52}	4	255	40	0,016
τ_{13}	5	765	25	0,007	τ_{53}	5	255	30	0,02
τ_{14}	6	765	85	0,008	τ_{54}	6	255	20	0,024
τ_{15}	7	765	85	0,009	τ_{55}	7	255	10	0,027
τ_{16}	3	765	65	0,004	τ_{56}	3	255	40	0,012
τ_{17}	4	765	45	0,005	τ_{57}	4	255	30	0,016
τ_{18}	5	765	25	0,007	τ_{58}	5	255	20	0,02
τ_{19}	6	765	5	0,008	τ_{59}	6	255	10	0,024
τ_{20}	7	1020	240	0,007	τ_{60}	7	255	0	0,027
τ_{21}	3	1020	220	0,003	τ_{61}	3	255	30	0,012
τ_{22}	4	1020	200	0,004	τ_{62}	4	255	20	0,016
τ_{23}	5	1020	180	0,005	τ_{63}	5	255	10	0,02
τ_{24}	6	1020	160	0,006	τ_{64}	6	255	0	0,024
τ_{25}	3	1020	120	0,003	τ_{65}	7	255	50	0,027
τ_{26}	3	1020	120	0,003	τ_{66}	4	255	10	0,016
τ_{27}	4	1020	100	0,004	τ_{67}	5	255	0	0,02
τ_{28}	5	1020	80	0,005	τ_{68}	6	255	50	0,024
τ_{29}	2	1020	60	0,002	τ_{69}	7	255	40	0,027
τ_{30}	3	1020	40	0,003	τ_{70}	8	255	30	0,031
τ_{31}	4	1020	40	0,004	τ_{71}	4	255	0	0,016
τ_{32}	5	1020	220	0,005	τ_{72}	5	255	50	0,02
τ_{33}	6	765	125	0,008	τ_{73}	6	255	40	0,024
τ_{34}	4	765	65	0,005	τ_{74}	7	255	30	0,027
τ_{35}	8	765	85	0,01	τ_{75}	8	255	20	0,031
τ_{36}	4	765	65	0,005	τ_{76}	4	255	50	0,016
τ_{37}	5	765	45	0,007	τ_{77}	5	255	40	0,02
τ_{38}	6	765	25	0,008	τ_{78}	6	255	30	0,024
τ_{39}	4	765	5	0,005	τ_{79}	7	255	20	0,027
τ_{40}	8	1020	240	0,008	τ_{80}	8	255	10	0,031
TOTAL								**5965**	**0,927**

FIGURE 3.8: Les nouvelles valeurs des périodes des tâches après l'application de l'heuristique A.

La Figure 3.9 présente un histogramme qui compare le coût de modification de périodes pour chaque tâche afin de réobtenir l'ordonnançabilité du système.

Les barres orange représentent le coût en appliquant la solution proposée par Wang et al. [2] et celles en bleue indiquent le coût après l'application de l'heuristique A. Nous remarquons que l'heuristique proposée ici donne une meilleure solution en termes de coût de modification des périodes des tâches.

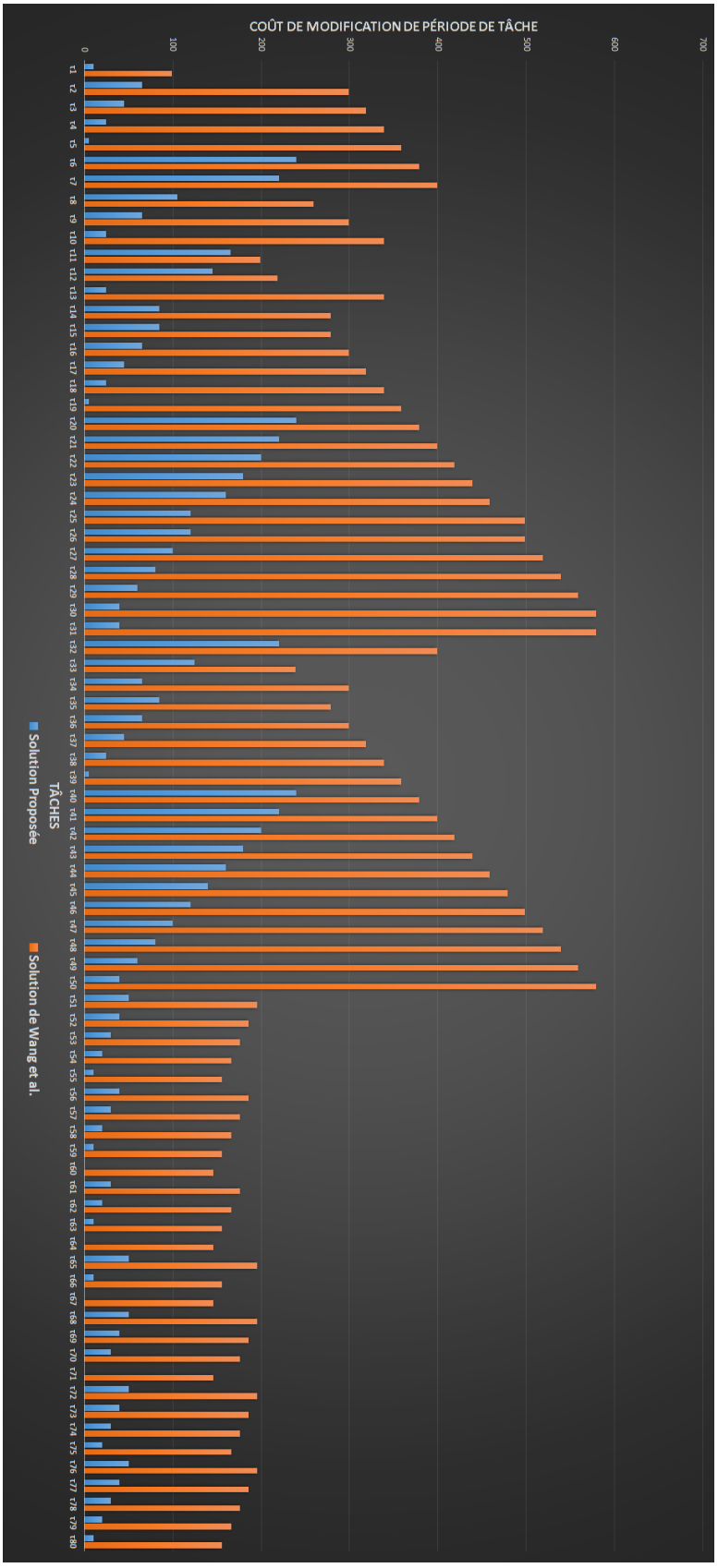

FIGURE 3.9: Comparaison en terme de coût de modification des périodes des tâches.

La comparaison avec des résultats optimaux fournis par le solveur CPLEX en résolvant $Pb1$, décrit dans la section 3.3.2, est importante pour situer les résultats de l'heuristique proposée. Après l'exécution de ces trois solutions, nous ajoutons le coût de la solution optimale dans la Figure 3.10 (barres en rouge).

- Le coût total en appliquant l'heuristique A : 5965
- Le coût total après l'utilisation de l'approche présentée dans [2] : 24550
- Le coût total fournit par CPLEX lorsqu'on résout $Pb1$: 4551

En d'autres termes, la solution proposée a un surcoût de 30% ($\frac{5965}{4551}$) par rapport à l'optimal, tandis que la solution selon Wang et al. [2] coûte 5,3 ($\frac{24550}{4551}$) fois plus que la solution optimale. Il est important de noter qu'il n'est pas possible d'implémenter un solveur sur une plate-forme embarquée alors que la solution proposée donne des résultats proches de l'optimal avec une possibilité d'implémentation.

Cas 2 : Modification des périodes avec prise en compte de la contrainte $T_{i_{max}}$:

Nous supposons maintenant que les 30 tâches ajoutées ont des périodes maximales très proches de leur période nominale (Figure 3.11). Pour les 50 tâches initiales, nous supposons que leur période maximale sont très élevées par rapport à leur période nominale. On ne considère pas donc une contrainte $T_{i_{max}}$ sur cet sous ensemble de tâches.

Selon la Figure 3.8 qui illustre les paramètres des tâches après l'application de l'heuristique A, nous remarquons que toutes les nouvelles périodes de 30 tâches ajoutées sont inférieurs à leur période maximale. Donc, notre heuristique reste valable dans ces conditions. Néanmoins la nouvelle période trouvée après l'application de la méthode de Wang et al. [2] (T_i=401) dépassent la limite définie par $T_{i_{max}}$. Dans ce cas, il s'agit de deux solutions possibles : i) calcul des nouvelles périodes en respectant $T_{i_{max}}$ ou ii) suppression de certaines tâches.

Wang et al. [2] proposent de modifier les périodes de tâches ou bien de supprimer certaines tâches afin de réduire le taux d'utilisation du processeur.

Nous nous intéressons au calcul des nouvelles périodes pour montrer le comportement de la stratégie proposée dans [2]. Pour satisfaire l'ordonnançabilité, il est nécessaire que :

$$\sum_{i=1}^{80} \frac{W_i}{T_i} \leq 1$$

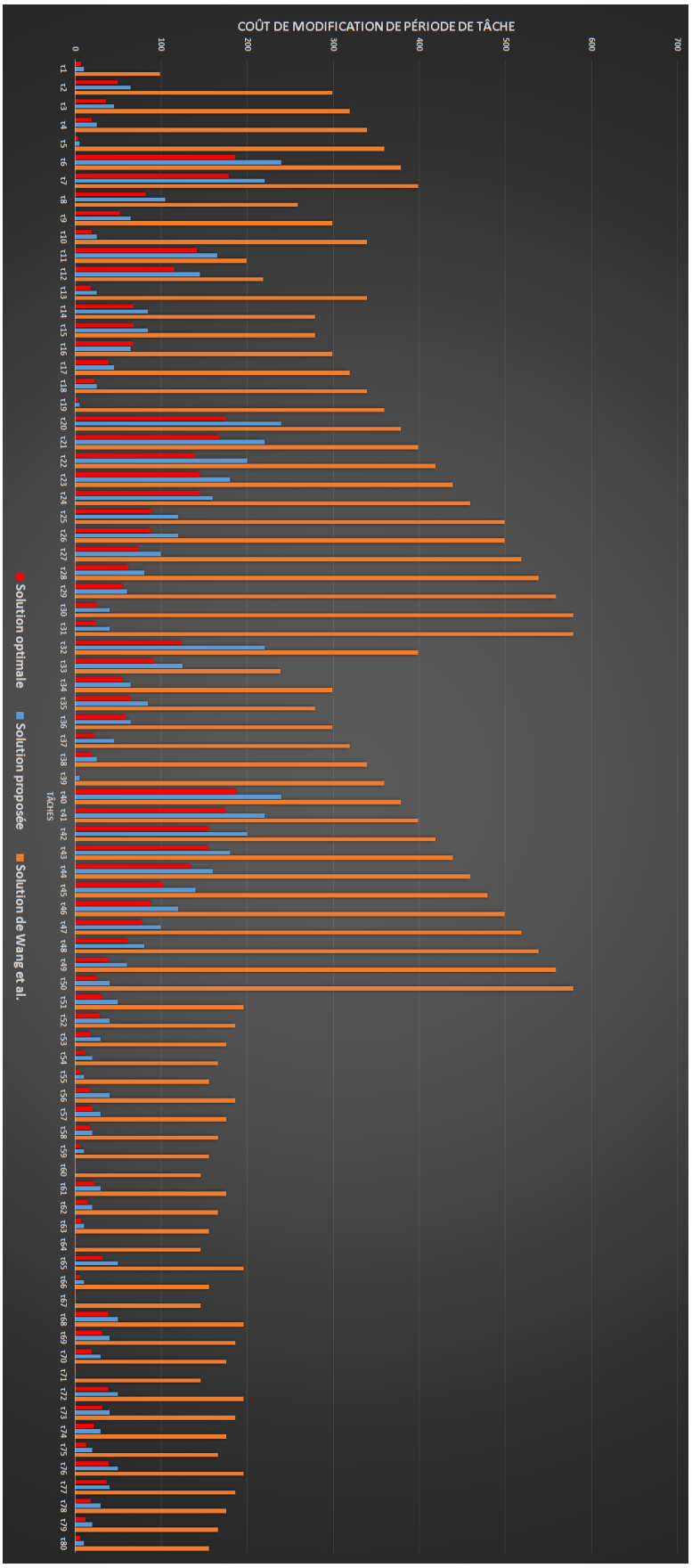

FIGURE 3.10: Comparaison de modification des périodes des tâches avec la solution optimale.

Tâche	W_i	T_i	T_{imax}	Tâche	W_i	T_i	T_{imax}
τ_{51}	3	205	300	τ_{66}	4	245	300
τ_{52}	4	215	260	τ_{67}	5	255	260
τ_{53}	5	225	260	τ_{68}	6	205	260
τ_{54}	6	235	275	τ_{69}	7	215	275
τ_{55}	7	245	300	τ_{70}	8	225	300
τ_{56}	3	215	255	τ_{71}	**4**	**255**	255
τ_{57}	4	225	361	τ_{72}	5	205	361
τ_{58}	5	235	275	τ_{73}	6	215	275
τ_{59}	6	245	255	τ_{74}	7	225	255
τ_{60}	7	255	256	τ_{75}	8	235	256
τ_{61}	3	225	302	τ_{76}	4	205	302
τ_{62}	4	235	260	τ_{77}	5	215	260
τ_{63}	5	245	298	τ_{78}	6	225	298
τ_{64}	6	255	283	τ_{79}	7	235	283
τ_{65}	7	205	300	τ_{80}	8	245	300

FIGURE 3.11: Périodes maximales pour les tâches ajoutées.

Puisque les 30 tâches ajoutées ont des périodes maximales proches de celles initiales, on ne va pas donc modifier leur période et on s'intéresse qu'à la modification des périodes de tâches initiales, i.e.,

$$\sum_{i=1}^{50} \frac{W_i}{T_i} + \sum_{i=51}^{80} \frac{W_i}{T_i} \leq 1$$

Ainsi,

$$\sum_{i=1}^{50} \frac{W_i}{T_i} \leq 1 - \sum_{i=51}^{80} \frac{W_i}{T_i}$$

d'ou, la nouvelle période pour les 50 tâches initiales est :

$$T' \geq \frac{\sum_{i=1}^{50} W_i}{1 - \sum_{i=51}^{80} \frac{W_i}{T_i}}$$

Comme les périodes sont des entiers, nous obtiendrons donc

$$T' = \left\lceil \frac{\sum_{i=1}^{50} W_i}{1 - \sum_{i=51}^{80} \frac{W_i}{T_i}} \right\rceil$$

Après avoir effectué une application numérique, la nouvelle période pour les 50 tâches initiales est 847. Le taux d'utilisation U_C devient donc 0.999. La Figure 3.12 représente les nouvelles périodes après l'application de la méthode de Wang et al. [2] en prenant en compte des contraintes sur les périodes.

Tâche	W_i	Nouvelle T_i	*CoûtModif* τ_i	U_{ti}	Tâche	W_i	Nouvelle T_i	*CoûtModif* τ_i	U_{ti}
τ_1	3	847	347	0,004	τ_{41}	4	847	47	0,005
τ_2	4	847	147	0,005	τ_{42}	5	847	27	0,006
τ_3	5	847	127	0,006	τ_{43}	6	847	7	0,007
τ_4	6	847	107	0,007	τ_{44}	7	847	13	0,008
τ_5	7	847	87	0,008	τ_{45}	8	847	33	0,009
τ_6	3	847	67	0,004	τ_{46}	4	847	53	0,005
τ_7	2	847	47	0,002	τ_{47}	5	847	73	0,006
τ_8	5	847	187	0,006	τ_{48}	3	847	93	0,004
τ_9	6	847	147	0,007	τ_{49}	3	847	113	0,004
τ_{10}	4	847	107	0,005	τ_{50}	4	847	133	0,005
τ_{11}	3	847	247	0,004	τ_{51}	3	205	0	0,015
τ_{12}	4	847	227	0,005	τ_{52}	4	215	0	0,019
τ_{13}	5	847	107	0,006	τ_{53}	5	225	0	0,022
τ_{14}	6	847	167	0,007	τ_{54}	6	235	0	0,026
τ_{15}	7	847	167	0,008	τ_{55}	7	245	0	0,029
τ_{16}	3	847	147	0,004	τ_{56}	3	215	0	0,014
τ_{17}	4	847	127	0,005	τ_{57}	4	225	0	0,018
τ_{18}	5	847	107	0,006	τ_{58}	5	235	0	0,021
τ_{19}	6	847	87	0,007	τ_{59}	6	245	0	0,024
τ_{20}	7	847	67	0,008	τ_{60}	7	255	0	0,027
τ_{21}	3	847	47	0,004	τ_{61}	3	225	0	0,013
τ_{22}	4	847	27	0,005	τ_{62}	4	235	0	0,017
τ_{23}	5	847	7	0,006	τ_{63}	5	245	0	0,02
τ_{24}	6	847	13	0,007	τ_{64}	6	255	0	0,024
τ_{25}	3	847	53	0,004	τ_{65}	7	205	0	0,034
τ_{26}	3	847	53	0,004	τ_{66}	4	245	0	0,016
τ_{27}	4	847	73	0,005	τ_{67}	5	255	0	0,02
τ_{28}	5	847	93	0,006	τ_{68}	6	205	0	0,029
τ_{29}	2	847	113	0,002	τ_{69}	7	215	0	0,033
τ_{30}	3	847	133	0,004	τ_{70}	8	225	0	0,036
τ_{31}	4	847	133	0,005	τ_{71}	4	255	0	0,016
τ_{32}	5	847	47	0,006	τ_{72}	5	205	0	0,024
τ_{33}	6	847	207	0,007	τ_{73}	6	215	0	0,028
τ_{34}	4	847	147	0,005	τ_{74}	7	225	0	0,031
τ_{35}	8	847	167	0,009	τ_{75}	8	235	0	0,034
τ_{36}	4	847	147	0,005	τ_{76}	4	205	0	0,02
τ_{37}	5	847	127	0,006	τ_{77}	5	215	0	0,023
τ_{38}	6	847	107	0,007	τ_{78}	6	225	0	0,027
τ_{39}	4	847	87	0,005	τ_{79}	7	235	0	0,03
τ_{40}	8	847	67	0,009	τ_{80}	8	245	0	0,033
TOTAL								**5260**	**0,999**

FIGURE 3.12: Nouvelles périodes après l'exécution avec prise en compte de la contrainte $T_{i_{max}}$.

En prenant en compte la contrainte $T_{i_{max}}$ sur les 30 nouvelles tâches, nous remarquons que la méthode de Wang et al. [2] a pu réduire le taux d'utilisation du processeur U_C, mais elle engendre un coût de modification supplémentaire important qui est égal à 5260 (Figure 3.12), alors que notre heuristique A n'a engendré aucun coût de modification supplémentaire.

Par ailleurs, puisque notre heuristique reste valable malgré la contrainte sur les périodes, donc aucune tâche ne sera pas supprimée. Néanmoins, Wang et al. [2] peuvent supprimer certaines tâches si la solution de modification de périodes n'a pas été appliquée.

Nous passons maintenant à l'évaluation de l'heuristique B. Nous comparons également cette heuristique avec celle proposée par Wang et al. [2]. Afin de simplifier le calcul, nous supposons que la fréquence nominale du cœur $F_n = 2Ghz$ et que ses points de fonctionnement sont continus dans l'ensemble $[2Ghz \dots 4Ghz]$.

$Reconf - Pack$ exécute la technique de diminution de temps d'exécution de tâches (heuristique B) et présente dans la Figure 3.13 la nouvelle valeur de WCETs après l'application de la méthode présentée dans [2]. Concernant les nouveaux WCETs obtenus en appliquant l'heuristique B, ils sont illustrés par la Figure 3.14.

Les deux solutions permettent de réduire le taux d'utilisation U_C à une valeur inférieure à 1 (voir les valeurs dans les Figures 3.13 et 3.14), mais en terme de coût de modification l'heuristique B coûte moins que celle proposée par Wang et al. [2]. Un histogramme de comparaison est représenté dans la Figure 3.15.

En gardant les mêmes couleurs des barres, nous remarquons que :

- Le coût total en appliquant l'heuristique B : 80
- Le coût total après l'utilisation de l'approche présentée dans [2] : 103
- Le coût total fournit par CPLEX est : 67

La solution proposée a un surcoût de 11% ($\frac{80}{67}$) par rapport à l'optimale, tandis que la solution de Wang et al. [2] a un surcoût presque de 16% ($\frac{103}{67}$) par rapport à l'optimale.

Tâche	Nouveau W_i	T_i	*CoûtModif* τ_i	U_{ti}	Tâche	Nouveau W_i	T_i	*CoûtModif* τ_i	U_{ti}
τ_1	5	500	2	0,01	τ_{41}	5	800	1	0,006
τ_2	5	700	1	0,007	τ_{42}	5	820	0	0,006
τ_3	5	720	0	0,007	τ_{43}	5	840	1	0,006
τ_4	5	740	1	0,007	τ_{44}	5	860	2	0,006
τ_5	5	760	2	0,007	τ_{45}	5	880	3	0,006
τ_6	5	780	2	0,006	τ_{46}	5	900	1	0,006
τ_7	5	800	3	0,006	τ_{47}	5	920	0	0,005
τ_8	5	660	0	0,008	τ_{48}	5	940	2	0,005
τ_9	5	700	1	0,007	τ_{49}	5	960	2	0,005
τ_{10}	5	740	1	0,007	τ_{50}	5	980	1	0,005
τ_{11}	5	600	2	0,008	τ_{51}	5	205	2	0,024
τ_{12}	5	620	1	0,008	τ_{52}	5	215	1	0,023
τ_{13}	5	740	0	0,007	τ_{53}	5	225	0	0,022
τ_{14}	5	680	1	0,007	τ_{54}	5	235	1	0,021
τ_{15}	5	680	2	0,007	τ_{55}	5	245	2	0,02
τ_{16}	5	700	2	0,007	τ_{56}	5	215	2	0,023
τ_{17}	5	720	1	0,007	τ_{57}	5	225	1	0,022
τ_{18}	5	740	0	0,007	τ_{58}	5	235	0	0,021
τ_{19}	5	760	1	0,007	τ_{59}	5	245	1	0,02
τ_{20}	5	780	2	0,006	τ_{60}	5	255	2	0,02
τ_{21}	5	800	2	0,006	τ_{61}	5	225	2	0,022
τ_{22}	5	820	1	0,006	τ_{62}	5	235	1	0,021
τ_{23}	5	840	0	0,006	τ_{63}	5	245	0	0,02
τ_{24}	5	860	1	0,006	τ_{64}	5	255	1	0,02
τ_{25}	5	900	2	0,006	τ_{65}	5	205	2	0,024
τ_{26}	5	900	2	0,006	τ_{66}	5	245	1	0,02
τ_{27}	5	920	1	0,005	τ_{67}	5	255	0	0,02
τ_{28}	5	940	0	0,005	τ_{68}	5	205	1	0,024
τ_{29}	5	960	3	0,005	τ_{69}	5	215	2	0,023
τ_{30}	5	980	2	0,005	τ_{70}	5	225	3	0,022
τ_{31}	5	980	1	0,005	τ_{71}	5	255	1	0,02
τ_{32}	5	800	0	0,006	τ_{72}	5	205	0	0,024
τ_{33}	5	640	1	0,008	τ_{73}	5	215	1	0,023
τ_{34}	5	700	1	0,007	τ_{74}	5	225	2	0,022
τ_{35}	5	680	3	0,007	τ_{75}	5	235	3	0,021
τ_{36}	5	700	1	0,007	τ_{76}	5	205	1	0,024
τ_{37}	5	720	0	0,007	τ_{77}	5	215	0	0,023
τ_{38}	5	740	1	0,007	τ_{78}	5	225	1	0,022
τ_{39}	5	760	1	0,007	τ_{79}	5	235	2	0,021
τ_{40}	5	780	3	0,006	τ_{80}	5	245	3	0,02
TOTAL								**103**	**0,974**

FIGURE 3.13: La nouvelle valeurs des WCETs des tâches après l'application de la méthode présentée dans [2].

Tâche	Nouveau W_i	T_i	*CoûtModif* τ_i	$U_{\tau i}$	Tâche	Nouveau W_i	T_i	*CoûtModif* τ_i	$U_{\tau i}$
τ_1	2	500	1	0,004	τ_{41}	3	800	1	0,004
τ_2	3	700	1	0,004	τ_{42}	4	820	1	0,005
τ_3	4	720	1	0,006	τ_{43}	5	840	1	0,006
τ_4	5	740	1	0,007	τ_{44}	6	860	1	0,007
τ_5	6	760	1	0,008	τ_{45}	7	880	1	0,008
τ_6	2	780	1	0,003	τ_{46}	3	900	1	0,003
τ_7	1	800	1	0,001	τ_{47}	4	920	1	0,004
τ_8	4	660	1	0,006	τ_{48}	2	940	1	0,002
τ_9	5	700	1	0,007	τ_{49}	2	960	1	0,002
τ_{10}	3	740	1	0,004	τ_{50}	3	980	1	0,003
τ_{11}	2	600	1	0,003	τ_{51}	2	205	1	0,01
τ_{12}	3	620	1	0,005	τ_{52}	3	215	1	0,014
τ_{13}	4	740	1	0,005	τ_{53}	4	225	1	0,018
τ_{14}	5	680	1	0,007	τ_{54}	5	235	1	0,021
τ_{15}	6	680	1	0,009	τ_{55}	6	245	1	0,024
τ_{16}	2	700	1	0,003	τ_{56}	2	215	1	0,009
τ_{17}	3	720	1	0,004	τ_{57}	3	225	1	0,013
τ_{18}	4	740	1	0,005	τ_{58}	4	235	1	0,017
τ_{19}	5	760	1	0,007	τ_{59}	5	245	1	0,02
τ_{20}	6	780	1	0,008	τ_{60}	6	255	1	0,024
τ_{21}	2	800	1	0,003	τ_{61}	2	225	1	0,009
τ_{22}	3	820	1	0,004	τ_{62}	3	235	1	0,013
τ_{23}	4	840	1	0,005	τ_{63}	4	245	1	0,016
τ_{24}	5	860	1	0,006	τ_{64}	5	255	1	0,02
τ_{25}	2	900	1	0,002	τ_{65}	6	205	1	0,029
τ_{26}	2	900	1	0,002	τ_{66}	3	245	1	0,012
τ_{27}	3	920	1	0,003	τ_{67}	4	255	1	0,016
τ_{28}	4	940	1	0,004	τ_{68}	5	205	1	0,024
τ_{29}	1	960	1	0,001	τ_{69}	6	215	1	0,028
τ_{30}	2	980	1	0,002	τ_{70}	7	225	1	0,031
τ_{31}	3	980	1	0,003	τ_{71}	3	**255**	1	0,012
τ_{32}	4	800	1	0,005	τ_{72}	4	205	1	0,02
τ_{33}	5	640	1	0,008	τ_{73}	5	215	1	0,023
τ_{34}	3	700	1	0,004	τ_{74}	6	225	1	0,027
τ_{35}	7	680	1	0,01	τ_{75}	7	235	1	0,03
τ_{36}	3	700	1	0,004	τ_{76}	3	205	1	0,015
τ_{37}	4	720	1	0,006	τ_{77}	4	215	1	0,019
τ_{38}	5	740	1	0,007	τ_{78}	5	225	1	0,022
τ_{39}	3	760	1	0,004	τ_{79}	6	235	1	0,026
τ_{40}	7	780	1	0,009	τ_{80}	7	245	1	0,029
TOTAL								**80**	**0,833**

FIGURE 3.14: Les nouvelles valeurs des WCETs des tâches après l'application de l'heuristique B.

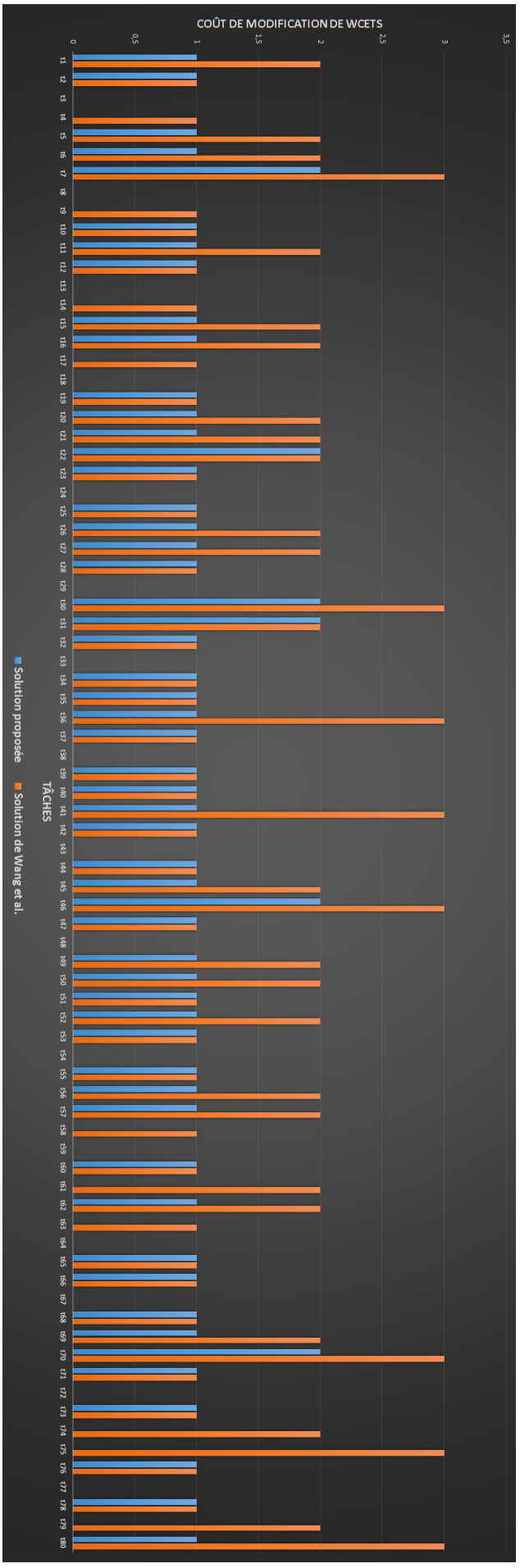

FIGURE 3.15: Comparaison en terme de coût de diminution de WCETs des tâches.

3.5.2 Génération aléatoire de tâches et calcul du gain moyen

Pour généraliser l'évaluation, nous comparons notre proposition avec l'algorithme de Wang et al. [2] sur des jeux de tâches générés aléatoirement par l'outil $Task-generator$ (Figure 3.16) qui est un module intégré dans le simulateur $Reconf-Pack$. Chaque jeu contient 100 tâches indépendantes comme suit :

- Chaque période T_i varie entre 200 et 1000,
- Chaque période maximale $T_{i_{max}}$ doit être entre T_i et $2 * T_i$
- Chaque WCET W_i doit être déterminé aléatoirement dans l'intervalle [1 ; 10]

Étant donné que le nombre de tâches est très élevé dans cette simulation, l'obtention d'une solution par le solveur prend beaucoup de temps et parfois un temps énorme. Pour cela, nous nous intéressons uniquement à comparer notre solution avec celle de Wang et al. [2]. La Table 3.3 présente le coût total de la modification de 50 jeux de tâches générés aléatoirement.

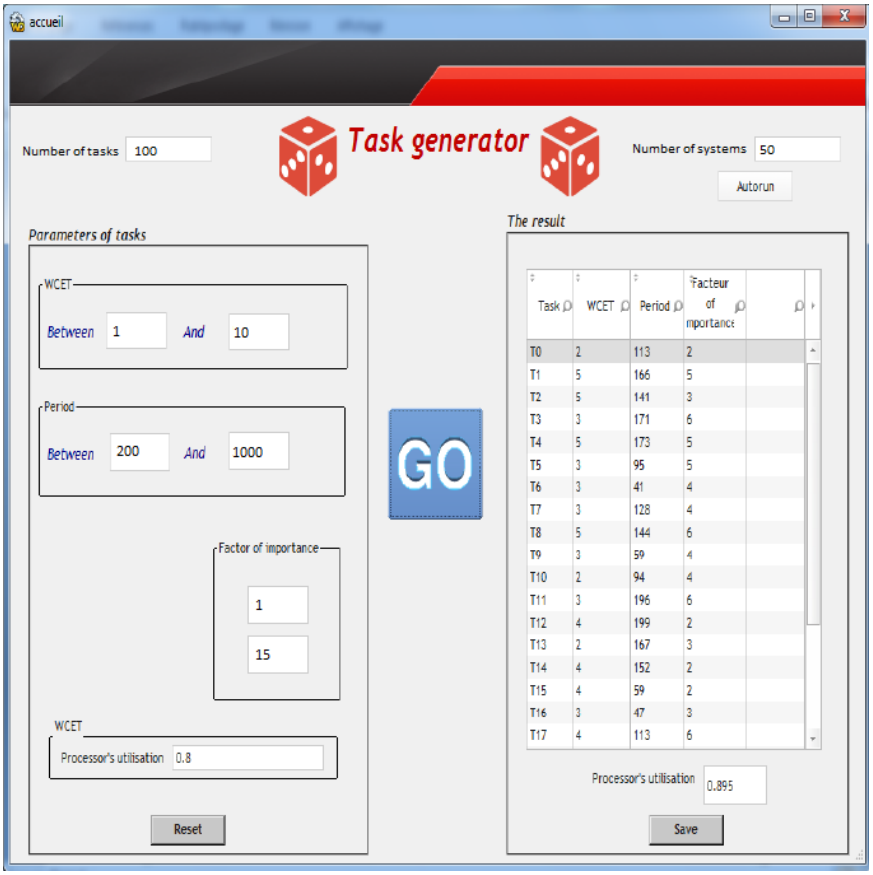

FIGURE 3.16: Interface graphique du module de génération aléatoire $Task-generator$.

TABLE 3.3: Coût total de modification des paramètres des tâches générées aléatoirement par $Task - genrator$

Système	Coût total de notre stratégie	Coût total de stratégie de [2]	**Système**	Coût total de notre stratégie	Coût total de stratégie [2]
Système 1	978	3524	**Système 26**	1054	3202
Système 2	630	1103	**Système 27**	11095	4257
Système 3	1917	2656	**Système 28**	11095	503
Système 4	1321	2465	**Système 29**	780	813
Système 5	313	644	**Système 30**	742	1354
Système 6	806	1828	**Système 31**	205	996
Système 7	428	395	**Système 32**	1220	999
Système 8	421	1407	**Système 33**	525	1003
Système 9	477	1200	**Système 34**	9306	9306
Système 10	647	1471	**Système 35**	1627	1697
Système 11	302	1065	**Système 36**	376	543
Système 12	307	1193	**Système 37**	327	801
Système 13	526	1229	**Système 38**	565	1438
Système 14	911	1449	**Système 39**	1136	2337
Système 15	1209	1209	**Système 40**	562	1578
Système 16	1171	1702	**Système 41**	177	228
Système 17	2554	2888	**Système 42**	495	256
Système 18	2951	2421	**Système 43**	483	2526
Système 19	1765	1889	**Système 44**	2048	4116
Système 20	1442	2862	**Système 45**	683	3526
Système 21	2522	2903	**Système 46**	1978	4127
Système 22	1065	2788	**Système 47**	1237	3699
Système 23	653	1805	**Système 48**	502	1293
Système 24	422	3535	**Système 49**	1451	1496
Système 25	1435	5093	**Système 50**	2821	6661
			Coût Total	79663	109479

Selon les résultats obtenus, notre stratégie minimise le coût de modification dans 88% des tests générés aléatoirement. De plus, elle offre 27% ($\frac{109479-79663}{109479} * 100$) de gain de coût de modification par rapport à l'approche proposée par Wang et al. [2]. 79663 étant le coût total (Table 3.3) selon notre approche et 109479 étant le coût total (Table 3.3) selon celle de Wang et al. [2].

Concernant les 12% des tests, nous avons remarqué que les tâches générées pour ces tests ont des périodes très diverses et qui ne sont pas du tout "similaires". Dans ce cas, la construction des packs engendre un coût plus important que celui trouvé lorsqu'on applique la solution de Wang et al. [2].

Après avoir montré le gain de notre stratégie par rapport à celle proposée par Wang et al. [2], il serait intéressant de concevoir un *Middleware* qui implémente cette stratégie et manager les paramètres des tâches afin de respecter les contraintes précitées. Cette étude conceptuelle est traitée dans la section suivante.

3.6 Conception d'un *Middleware* reconfigurable

Étant donné qu'un système d'exploitation classique n'est pas prévu pour supporter la reconfiguration au sens où nous l'avons défini, nous proposons une couche intermédiaire $Reconf - Middleware$ [97] qui va jouer le rôle d'intergiciel. Cette couche implémente la stratégie proposée dans ce chapitre et elle va être en interaction avec le noyau de RTLinux [98]. Elle va gérer les scénarios de reconfiguration et appliquer les solutions adéquates pour rendre le système faisable après un scénario de reconfiguration. Cette couche est représentée par la Figure 3.17.

Une modélisation graphique basée sur UML a été choisie afin d'avoir une représentation claire des différents éléments du modèle général du système [99] ainsi que les éléments du *Middleware* $Reconf - Middleware$. Le modèle inclut les tâches applicatives, tous les composants matériels et logiciels du système ainsi que les composants pour gérer les reconfigurations. Nous nous appuyons plus particulièrement sur une représentation à l'aide d'un diagramme de classes UML, tel que le montre la Figure 3.18. Nous avons choisi de décomposer la modélisation en quatre parties correspondante chacune à une couche de description différente : (la *couche application*, la *couche OS*, la *couche de Middleware*

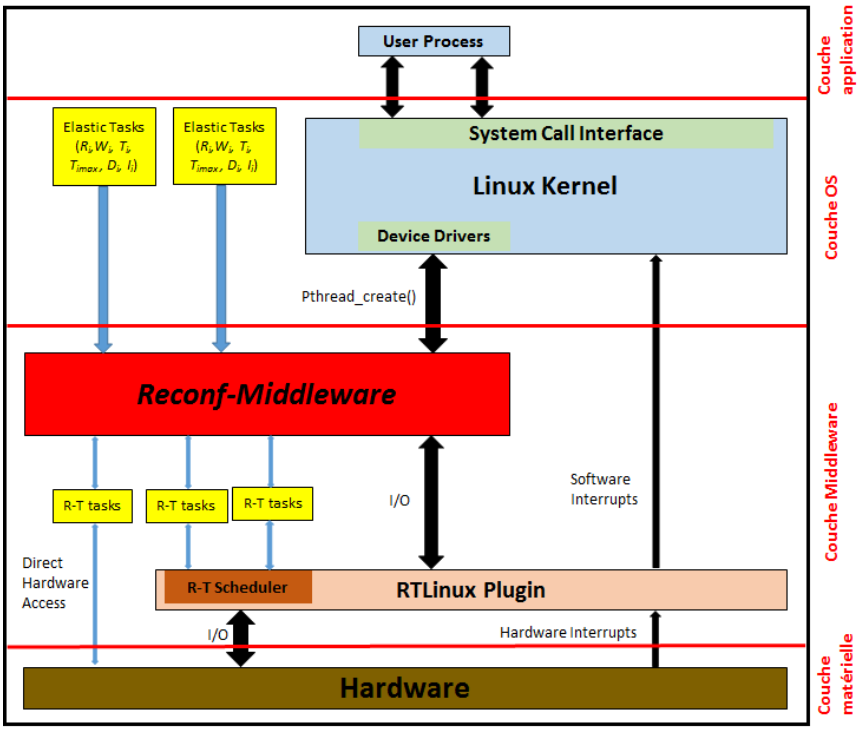

FIGURE 3.17: Localisation de la couche intermédiaire *Reconf – Middleware*.

et la *couche matérielle*). Ceci permet d'exprimer l'indépendance de l'application vis-à-vis de l'architecture.

La couche **application** est basée essentiellement sur la description des tâches applicatives, représentées par la classe PeriodicTasks. C'est la couche utilisée par le concepteur "logiciel" pour représenter son application sans se préoccuper de l'architecture matérielle. Ainsi, à ce niveau de description, n'existent que des informations liées aux traitements à réaliser.

La couche **OS** représente le système d'exploitation. L'ordonnancement de l'ensemble de tâches est effectué dans cette couche. Elle identifie également un ensemble de services de l'OS relatifs aux tâches et à la gestion de l'architecture matérielle.

La couche **middleware** représente les différentes solutions qui vont être appliquées par le gestionnaire des reconfigurations (classe ReconfManager). Un événement de reconfiguration est représenté par la classe Reconfiguration. Cette couche assure la modification des paramètres des tâches afin d'avoir un système faisable.

La couche **matérielle** représente les entités matérielles du système présentes physiquement dans la plate-forme. Cette couche englobe les cibles de traitement (classe Processor)

qui supportent l'exécution des tâches, les cibles de stockage (classe Memory) des données manipulées et enfin l'unité de stockage de l'énergie (classe Battery).

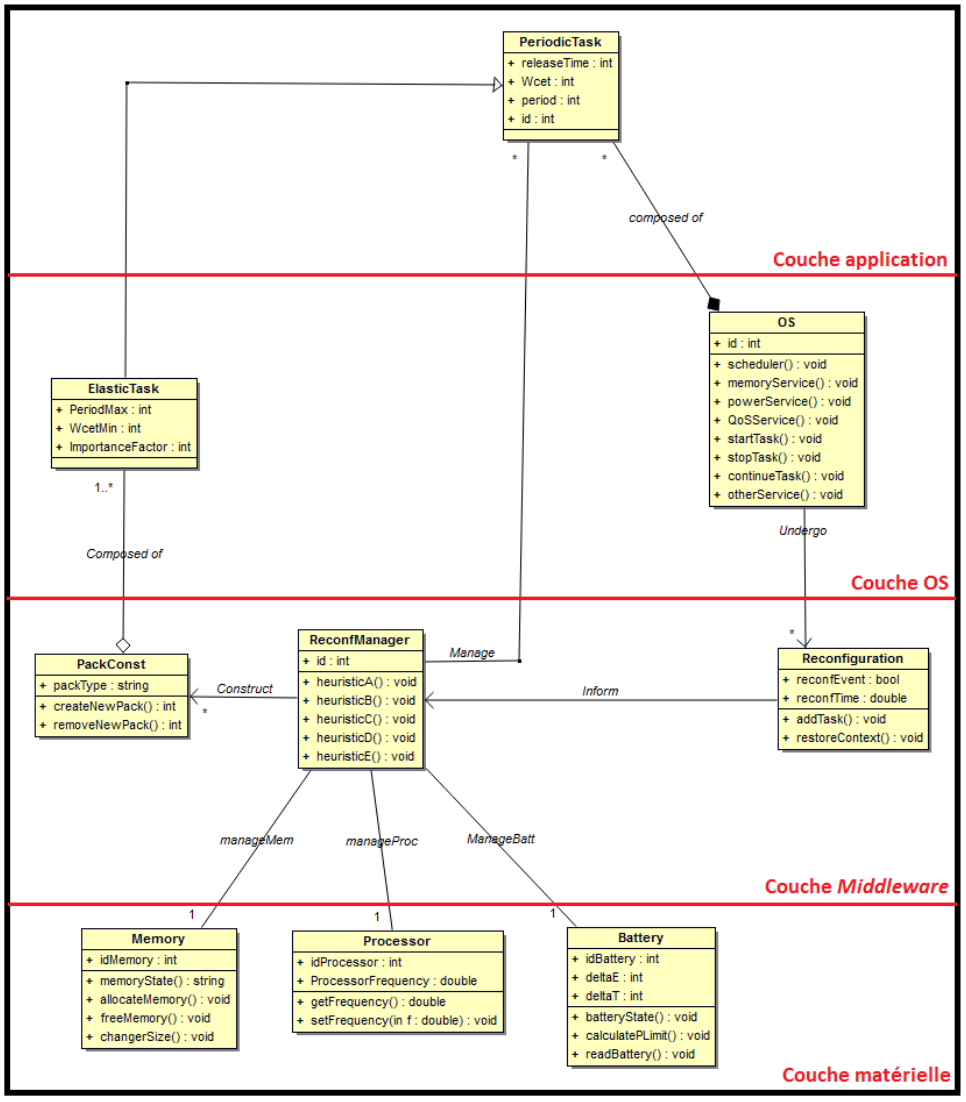

FIGURE 3.18: Diagramme de classes du modèle système général avec représentation des couches application, reconfiguration et matérielle ainsi que les interactions.

Concernant l'implémentation du middleware, nous verrons en perspectives que cette partie pourrait venir compléter notre étude et permettre d'avoir un RTOS embarqué qui reconfigure ses tâches pour respecter les contraintes précitées.

3.7 Conclusion

Dans ce chapitre, nous avons traité la problématique de l'ordonnancement de tâches temps-réel sous contrainte énergétique pour les architectures mono-cœur reconfigurables

et alimentées par batterie. Les modèles de tâches et de la consommation d'énergie ont été décrit afin de formaliser les problèmes. L'objectif était de proposer des solutions ainsi qu'une stratégie pour assurer l'ordonnancement des tâches du système après des scénarios de reconfiguration qui peuvent violer les contraintes temps-réel et énergétique. Notre stratégie permet d'assurer l'ordonnancement des tâches du système en appliquant une ou plusieurs heuristiques, et en adaptant les caractéristiques des tâches. Les résultats montrent que notre approche permet de réduire le coût de modification des paramètres par rapport à l'approche présentée dans [2]. Les résultats expérimentaux sur notre stratégie montrent qu'elle offre 27% de gain en terme de coût de modification par rapport à l'approche proposée par Wang et al. [2]. Par ailleurs, une comparaison avec une solution optimale a été réalisée sur le même cas d'étude présenté dans [2] et nous avons montré que notre approche a un surcoût de 30% par rapport o l'optimal. Ces travaux ont été prolongés en proposant un *Middleware* qui implémente ces heuristiques et qui peut être intégré au sein de différents systèmes d'exploitations. Le chapitre suivant va adresser la problématique de l'ordonnancement de tâches dépendantes sur des architectures multi-cœurs hétérogènes supportant l'exécution d'applications temps-réel. Une nouvelle contrainte sera traitée qui est la contrainte de communication. Après des scénarios de reconfiguration, cette nouvelle contrainte vérifie que le support de communication assurant le transfert des messages entre les tâches, est toujours capable d'assurer le transfert.

Chapitre 4

Contribution au placement et à l'ordonnancement des tâches temps-réel sous contraintes de communication et d'énergie pour les architectures multi-cœurs hétérogènes

4.1 Introduction

Dans le chapitre précédent, nous avons présenté une stratégie d'ordonnancement temps-réel sous contrainte énergétique pour les architectures mono-cœur reconfigurables. Dans ce nouveau chapitre, nous étendons les propositions précédentes pour adresser les architectures multi-cœurs. Pour cela, nous présentons d'abord les modèles matériel et logiciel auxquels nous nous intéressons puis nous considérons des tâches périodiques et dépendantes d'autres tâches au travers d'échanges de messages sur un réseau de communication [100]. Un réseau de communication peut être une liaison *point-to-point* (P2P), une liaison *multi-point* (bus), ou un *Network-on-Chip* (NoC). Dans le cas d'une architecture multi-cœurs homogène, une tâche est par défaut exécutable sur tous les cœurs, c'est-à-dire

qu'elle peut être placée sur n'importe quel cœur. Cependant dans le cas d'architecture multi-cœurs hétérogènes où les cœurs ne sont pas identiques, une tâche ne peut être exécutée que sur un ensemble de cœurs spécifiques qui est bien définit par le concepteur de l'application. Ainsi, une architecture homogène n'est donc qu'un cas particulier d'architectures hétérogènes. Par conséquent, nous abordons principalement dans ce chapitre le cas d'une architecture multi-cœurs hétérogènes où les cœurs peuvent exécuter de nouvelles tâches. Le problème se pose généralement au niveau du placement des tâches soit à la configuration initiale soit après l'ajout des nouvelles tâches suite à un événement de reconfiguration. Certains travaux [69–72] considèrent que la consommation énergétique des systèmes multi-cœurs hétérogènes est fortement dépendante du placement des tâches. Huang et al. [69, 70] supposent que la consommation énergétique est proportionnelle à la distance entre les cœurs. Autrement dit, plus les tâches dépendantes sont placées sur des cœurs éloignés, plus la consommation d'énergie augmente. L'idée sous jacente est que cette consommation dépend fortement des communications entre tâches. Par conséquent, l'objectif de ce chapitre consiste à proposer une stratégie de placement et d'ordonnancement des tâches en minimisant le coût total de communication afin de minimiser la consommation énergétique. Cette stratégie place les tâches en utilisant une nouvelle méthode de *Clustering* (Section 4.4.1) qui est basée sur l'élasticité des tâches. Notre stratégie vise dans un premier temps à placer sur le même cœur, les tâches qui échangent beaucoup de données lorsque cela est possible ou de réduire la distance entre ces tâches si le placement sur le même cœur n'est pas possible. Dans un deuxième temps, elle vérifie si les trois contraintes (temps-réel, communication et énergétique) sont respectées. S'il y a au moins une contrainte violée après un événement d'ajout de tâches, la stratégie propose d'appliquer des solutions afin de réobtenir la faisabilité du système. En plus des heuristiques A et E, présentées dans le chapitre précédent, nous proposerons dans ce chapitre deux autres heuristiques F et H.

Dans les sections suivantes, nous présentons tout d'abord la formalisation mathématique du problème. Nous indiquons ensuite les différentes contraintes liées à ce problème, notamment la contrainte de placement des tâches en minimisant le coût de communication. Nous détaillerons, dans la Section 4.4, les heuristiques proposées qui supporteront la stratégie de placement et d'ordonnancement de tâches qui peut être exécutée après chaque reconfiguration subit par le système. Finalement, une étude comparative sera réalisée dans la dernière section afin de montrer que notre approche permet d'améliorer

les résultats de l'état de l'art sur ce sujet.

4.2 Formalisation du système basé sur l'architecture multi-cœurs hétérogènes

Cette section est consacrée à la formalisation des modèles matériel/logiciel d'un système *RTSys* basé sur l'architecture muti-cœurs hétérogènes supportant des tâches qui communiquent entres elles via un médium de communication. Dans cette section, seront représentés les modèles d'architecture, de communication et de consommation énergétique totale. Concernant le modèle de tâches, nous conservons celui du chapitre précédant en ajoutant une nouvelle condition sur les WCETs de tâches pour distinguer les valeurs de WCET d'un cœur à un autre (architecture hétérogène).

4.2.1 Modèle d'architecture

Nous supposons que la plate-forme utilisée est composée par un ensemble Γ de p cœurs hétérogènes Γ=$\{C_1, C_2, \ldots, C_p\}$. En se basant sur l'approche de Chetto et al. [101], nous supposons que chaque cœur C_k ($k \in [1 \ldots p]$) ordonnance localement les tâches qui lui ont été affectées avec l'algorithme EDF. Nous considérons que chaque cœur peut voir augmenter, dynamiquement, son nombre de tâches à exécuter. Étant donné, que nous considérons une plate-forme multi-cœurs hétérogènes, nous supposons donc que chaque tâche τ_i peut être exécutée par un ou plusieurs cœurs spécifiques en fonction de la conception de l'application. Par conséquent, nous définissons une matrice γ qui représente les possibilités de placement pour chaque tâche.

$$\gamma_{i,k} = \begin{cases} 1 & \text{si } \tau_i \text{ peut être exécutée sur le cœur } C_k. \\ 0 & \text{sinon.} \end{cases}$$

Dans le cas d'une architecture multi-cœurs homogènes : où les cœurs sont identiques, nous n'avons pas besoin de définir une matrice de possibilités de placement γ, car chaque tâche peut être exécutée sur n'importe quel cœur.

Pour vérifier que toutes les tâches soient placées sur les différents cœurs, nous définissons la matrice de placement H telle que :

$$H_{i,k} = \begin{cases} 1 & \text{si la tâche } \tau_i \text{ est placée sur le cœur } C_k. \\ 0 & \text{sinon.} \end{cases} \tag{4.1}$$

Par ailleurs, une tâche ne peut être placée que sur l'un des cœurs spécifiques. Cette condition est formulée dans la contrainte :

$$\sum_{k=1}^{p} H_{i,k} * \gamma_{i,k} = 1 \qquad \forall i \in [1 \dots N] \tag{4.2}$$

Dans le cas d'une architecture multi-cœurs homogènes : cette condition peut être comme suit :

$$\sum_{k=1}^{p} H_{i,k} = 1 \qquad \forall i \in [1 \dots N] \tag{4.3}$$

4.2.2 Modèle de communication

Étant donné que les tâches sont dépendantes et peuvent échanger des messages entre elles, nous dénotons par $M_{i,j}$ le message périodique envoyé de τ_i à τ_j. En se basant sur les travaux de Bui et al. [102], chaque message $M_{i,j}$ est caractérisé par :

- Une période $TM_{i,j}$: Le temps régulier d'arrivée,
- Le pire temps de transmission (*Worst Case Transmission Time WCTT*) $WM_{i,j}$: Le temps nécessaire pour transmettre un message,
- Une échéance $DM_{i,j}$: C'est le temps limite absolu à ne pas dépasser,
- Une taille $SM_{i,j}$: La taille relative du message en bit.

Nous notons que chaque message $M_{i,j}$ hérite de la même valeur de période que la tâche émettrice T_i, i.e, $TM_{i,j} = T_i$. En outre, si la période d'une tâche est modifiée alors les périodes de leurs messages associés seront également changées implicitement.

La communication entre deux cœurs est assurée par un médium de communication qui peut être un bus, un NoC, etc. Pour transférer un message de tâche τ_i du cœur C_k à

la tâche τ_j du cœur C_l sur un médium de communication, Sandstrom et al. dans [103] proposent une méthode pour fragmenter ce message en plusieurs paquets. Cette méthode a été améliorée par Ricardo Santos et al. [104] en proposant une nouvelle façon de construire les paquets en temps-réel et qui est applicable aux ensembles des messages/signaux issus de l'industrie automobiles. En se basant sur ces travaux, nous supposons que chaque message $M_{i,j}$ peut être véhiculé sous forme de plusieurs paquets sur le médium de communication et nous n'adressons pas cette partie dans nos travaux.

Par ailleurs, chaque communication entre deux cœurs C_k et C_l se caractérise par un coût qui dépend de la distance entre ceux-ci. On note par $Cost_{C_k,C_l}$ ce coût. En d'autre terme, $Cost_{C_k,C_l}$ est le coût (e.g énergie) pour envoyer un bit du cœur C_k au cœur C_l.

Similairement aux travaux [69, 70], nous supposons que l'énergie consommée par la communication est proportionnelle à la distance entre cœurs. Ainsi, le coût de communication entre deux cœurs est le produit de la distance par la taille des données échangées. Par conséquent, le coût de communication total (en unité de coût) est calculé comme suit :

$$TotalCost = \sum_{k=1}^{p}\sum_{l=1}^{p}\sum_{i=1}^{N}\sum_{j=1}^{N} SM_{i,j} * Cost_{C_k,C_l} * H_{i,k} * H_{j,l} \tag{4.4}$$

4.2.3 Modèles d'utilisation du processeur et du support de communication

Le taux d'utilisation d'un cœur C_k est calculé comme suit :

$$U_{C_k} = \sum_{i=1}^{N} \frac{W_i}{T_i} * H_{i,k}; \forall k \in [1..p] \tag{4.5}$$

C_k est faisable si et seulement si :

$$U_{C_k} \leq 1; \forall k \in [1 \ldots p] \tag{4.6}$$

Puisque les messages échangés entre les cœurs C_k et C_l sont considérés préemptibles (Section 4.2.2), nous supposons que le médium de communication entre ces cœurs est un processeur virtuel [17, 102] pour pouvoir calculer son facteur d'utilisation :

$$U_{Com}(C_k, C_l) = \sum_{i=1}^{N}\sum_{j=1}^{N} \frac{WM_{i,j}}{TM_{i,j}} * H_{i,k} * H_{j,l}; \forall k, l \in [1..p] \tag{4.7}$$

Selon l'approche de Khemaissia et al. [30], la communication entre deux cœurs C_k et C_l est faisable si l'Eq. 4.8 est vérifiée.

$$U_{Com}(C_k, C_l) \leq 1; \forall k, l \in [1 \dots p]^2 \tag{4.8}$$

L'Eq. 4.8 qui vérifie la contrainte de communication, sera notée dans la suite du mémoire par **CommConst**.

4.2.4 Modèle de consommation d'énergie totale

Nous rappelons que selon [2, 92, 105] la puissance consommée par le système est proportionnelle à l'utilisation du processeur. Puisque notre système est multi-cœurs, la puissance d'un seul cœur C_k ($k \in [1 \dots p]$) est calculée comme suit :

$$P_{C_k} = k1 * U_{C_k}^2; \forall k \in [1 \dots p] \tag{4.9}$$

Par conséquent, la puissance consommée par toutes les tâches qui s'exécutent sur les p cœurs est calculée par :

$$P_{Proc} = \sum_{k=1}^{p} P_{C_k} \tag{4.10}$$

Par ailleurs, nous pouvons définir la puissance consommée par l'ensemble des communications comme suit :

$$P_{Com} = k2 * \sum_{k=1}^{p} \sum_{l=1}^{p} U_{Com}^2(C_k, C_l) \tag{4.11}$$

Ainsi, la puissance totale P_{Total} du $RTSys$ est la somme de la puissance consommée par les cœurs et celle consommée par le médium de communication :

$$P_{Total} = P_{Proc} + P_{Com} \tag{4.12}$$

En détaillant cette formule (Eq. 4.12), nous trouvons :

$$P_{Total} = k1 * \sum_{k=1}^{p} P_{C_k} + k2 * \sum_{k=1}^{p} \sum_{l=1}^{p} U_{Com}^2(C_k, C_l)$$

Ainsi,

$$P_{Total} = k1 * \sum_{k=1}^{p} \left(\sum_{i=1}^{N} \frac{W_i}{T_i} * H_{i,k} \right)^2 + k2 * \sum_{k=1}^{p} \sum_{l=1}^{p} \left(\sum_{i=1}^{N} \sum_{j=1}^{N} \frac{WM_{i,j}}{TM_{i,j}} * H_{i,k} * H_{j,l} \right)^2$$

Identiquement comme nous l'avons défini dans la section 3.2.3, P_{Limit} peut être calculée en utilisant les deux entrées suivantes : $E(t)$ et $t_{recharge}$. Ainsi, à un instant donné t et pour garantir que le système peut s'exécuter correctement jusqu'à la prochaine recharge en gardant la puissance consommée actuelle, il est nécessaire de vérifier que :

$$P_{Total}(t) \leq P_{Limit}(t) \tag{4.13}$$

Eq. 4.13 désigne la contrainte énergétique du $RTSys$.

4.3 Formalisation du problème de reconfiguration

Le problème abordé dans ce chapitre s'articule autour du placement des nouvelles tâches sur les cœurs après tout scénario de reconfiguration. Un placement optimal est considéré comme une allocation qui minimise le coût de la communication entre les cœurs, de sorte que les contraintes de temps-réel, communication et énergétique soient satisfaites. Nous modélisons ce problème avec une formulation ILP.

4.3.1 Problème de placement de tâches

Nous supposons qu'à l'instant t_i, $RTSys$ est composé par : $\Gamma(t_i) = \{C_1, C_2, \ldots, C_p\}$ et $\Pi(t_i) = \{\tau_1, \tau_2, \ldots, \tau_{n_1}\}$. $RTSys$ est dit faisable si et seulement si, il satisfait à la fois les trois contraintes (temps-réel, communication et énergétique). Nous supposons que $RTSys$ subit, à l'instant t_k ($t_k > t_i$), un scénario de reconfiguration ajoutant n_2 tâches. Le nouvel ensemble de tâches est $\Pi(t_k) = \{\tau_1, \tau_2, \ldots, \tau_{n_1}, \tau_{n_1+1}, \ldots, \tau_N\}$, avec $N = n_1 + n_2$. Aussi, chaque nouvelle tâche peut avoir une dépendance avec des anciennes/nouvelles tâches. Par conséquent, certains nouveaux messages peuvent être rajoutés sur le support de communication. Nous notons cette reconfiguration dans le reste du chapitre par $RECONF$. Nous considérons que la configuration initiale à l'instant t_0 est semblable à un scénario de reconfiguration tel que : $\Pi(t_0^-) = \{\varnothing\}$ et $\Pi(t_0) \neq \{\varnothing\}$.

Après un tel scénario (configuration ou reconfiguration), l'utilisation du processeur ainsi que l'utilisation du médium de communication augmentent et $RTSys$ risque d'être non faisable. L'objectif est donc de placer les nouvelles tâches tout en gardant la faisabilité du $RTSys$ après $RE\text{-}CONF$ tout en minimisant le coût de communication entre les cœurs. Nous formalisons ce problème sous forme d'une fonction à minimiser :

$$RE \begin{cases} \text{Minimiser } TotalCost \\ t.q. \\ \quad \sum_{k=1}^{p} H_{i,k} * \gamma_{i,k} = 1,\ \forall\ i \in [1 \dots N] \quad (Eq.4.3) \\ \quad U_{C_k} \leq 1,\ \forall\ k \in [1 \dots p] \quad (Eq.4.6) \\ \quad U_{Com}(C_k, C_l) \leq 1\ \ \forall\ k, l \in [1 \dots p]^2 \quad (Eq.4.8) \\ \quad P_{Total}(t) \leq P_{Limit}(t) \quad (Eq.4.13) \end{cases}$$

Avec, $TotalCost$ désigne le coût total de l'ensemble des communications dans le système, Eq. 4.3 permet de garantir que chaque tâche est placée sur l'un de ses cœurs appropriés, Eq. 4.6 vérifie que chaque cœur est ordonnançable, Eq. 4.8 vérifie la faisabilité du médium de communication. La dernière contrainte (Eq. 4.13) assure que $RTSys$ peut s'exécuter correctement jusqu'à la prochaine recharge dans cette configuration là.

Dans le cas où l'une de ces contraintes n'est pas respectée, la résolution du problème RE ne donne aucune solution (i.e., les tâches ne sont pas placées). Nous proposons donc d'enlever une contrainte (celle qui n'est pas vérifiée) et de résoudre à nouveau RE afin de trouver au moins un placement initial de tâches. Nous appliquons, par la suite, l'heuristique adéquate pour tenter de satisfaire la faisabilité du système.

Dans l'Eq. 4.13 du problème RE, il s'agit de deux facteurs $k1$ et $k2$ qui sont dépendantes de l'architecture matérielle utilisée. $k1$ est un facteur utilisé pour calculer la consommation de puissance des processeurs (Eq. 4.9) et celui $k2$ est utilisé pour calculer la consommation du support de communication (Eq. 4.11). Cela veut dire que ces deux

paramètres ne sont généralement pas identiques. Néanmoins, pour effectuer des simulations il faudrait les initialiser afin de pouvoir appliquer les formules qui estiment la consommation énergétique. Pour simplifier donc le calcul, nous supposons d'initialiser $k1$ et $k2$ à 1 ($k1$=$k2$=1).

4.3.2 Problème de contrainte énergétique totale

Suite à l'événement *RE-CONF*, de nouvelles tâches/messages peuvent être ajoutés. Par conséquent, la puissance totale consommée augmente et $RTSys$ peut ne pas être capable de poursuivre son exécution jusqu'à la prochaine recharge à cause de la violation de la contrainte énergétique. Afin d'appliquer la stratégie des packs pour réobtenir la faisabilité du $RTSys$, nous suggérons de traiter le cœur le plus chargé d'abord. Autrement dit, nous allongeons les périodes des tâches qui s'exécutent sur le cœur qui possède le taux d'utilisation le plus élevé, noté C_{high} ($high \in [1 \dots p]$), jusqu'à la satisfaction de la contrainte énergétique. Nous formalisons ce problème par :

$$Pb5 \begin{cases} \text{Minimiser } \sum_{i=1}^{N} \Delta T_i * H_{i,high} \\ t.q. \\ \quad \sum_{i=1}^{N} \frac{W_i}{T_i + \Delta T_i} * H_{i,high} \leq 1 \\ \quad T_i + \Delta T_i \leq T_{i_{max}}, \ \forall \ i \in [1 \dots N] \\ \quad \sum_{k=1}^{p} \left(\sum_{i=1}^{N} \frac{W_i}{T_i} * H_{i,k} \right)^2 + \sum_{k=1}^{p} \sum_{l=1}^{p} \left(\sum_{i=1}^{N} \sum_{j=1}^{N} \frac{WM_{i,j}}{TM_{i,j}} * H_{i,k} * H_{j,l} \right)^2 \leq P_{Limit} \end{cases} \quad (Eq.4.13)$$

Après l'allongement des périodes de tâches qui s'exécutent sur C_{high}, l'utilisation du processeur va être diminuée et par la suite la contrainte énergétique peut être satisfaite. Si elle est encore non vérifiée, nous appliquons la même stratégie sur les tâches du prochain cœur le plus chargé.

4.3.3 Problème de contrainte de communication

Cette sous section traite le problème de contrainte de communication après l'ajout de message(s) sur le médium de communication. Après cet événement, $U_{Com}(C_k, C_l)$ ($\forall k, l \in$

$[1 \dots p]$) peut dépasser 1 et par conséquent les messages ne satisfont pas leur échéance. Pour résoudre ce problème, nous proposons d'adapter les périodes des messages, c'est-à-dire que nous allongeons les périodes des messages afin de baisser la charge du médium de communication. L'idée est formalisée comme suit :

$\forall\ k, l \in [1..p] \mid U_{Com}(C_k, C_l) > 1$

$$Pb6 = \begin{cases} \text{Minimiser} \sum_{i=1}^{N} \sum_{j=1}^{N} \Delta TM_{i,j} \\ t.q. \\ \quad \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{WM_{i,j}}{TM_{i,j} + \Delta TM_{i,j}} * H_{i,k} * H_{i,l} \leq 1 \\ \quad TM_{i,j} + \Delta TM_{i,j} \leq T_{i_{max}},\ \forall\ i, j \in [1 \dots N] \end{cases}$$

Avec $\Delta TM_{i,j}$ un entier à ajouter à la période du message $M_{i,j}$ afin de baisser l'utilisation du médium de communication.

Nous pouvons résoudre $Pb5$ et $Pb6$ par un solveur CPLEX [81] qui peut fournir une solution optimale, mais il n'est pas raisonnable d'implémenter un solveur dans un système embarqué puisqu'il peut prendre beaucoup temps pour produire des solutions. Pour cela, nous présentons dans la section suivante des heuristiques efficaces sur un système temps-réel embarqué.

4.4 Contribution et heuristiques de placement et d'ordonnancement proposées

Dans cette section, nous présentons une stratégie de placement de tâches basée sur le concept du *Task Clustering* qui a comme rôle de placer les tâches communicantes sur les cœurs appropriés en réduisant la distance entre elles. Elle évalue en même temps la faisabilité du $RTSys$ en vérifiant les trois contraintes une par une. S'il existe une violation de contrainte, elle propose de modifier les paramètres de tâches/messages. Si $RTSys$ reste encore non faisable malgré la modification des paramètres, la stratégie propose de supprimer certaines tâches jugées non importantes.

4.4.1 Stratégie de placement de tâches

Après un événement de configuration initiale ou de reconfiguration $RE\text{-}CONF$, les nouvelles tâches doivent être placées et ordonnancées en respectant les contraintes associées (**RTConst**, **EgConst** et **CommConst**). En effet, nous proposons une stratégie de placement de tâches qui a comme objectif de minimiser les communications entre cœurs en satisfaisant les trois contraintes précitées.

Principe de la stratégie :

La stratégie proposée dans ce chapitre résout le problème de placement de tâches avec un coût de communication minimal. Pour minimiser les communications entre cœurs, elle place les tâches qui communiquent fréquemment sur le même cœur ou bien sur des cœurs proches. L'idée consiste à regrouper les tâches communicantes en différents sous-ensembles appelés *Cluster* (noté **CL**). Nous calculons par la suite la quantité de données transférées entre les tâches pour chaque *Cluster* afin de donner la priorité de placement à celui dont la quantité de données à transférer est la plus grande.

L'objectif de la stratégie proposée dans ce chapitre, est de s'assurer des points suivants :

- Chaque tâche doit être placée sur un et un seul cœur qui peut l'exécuter,
- La charge de chaque cœur ne doit pas dépasser sa charge maximale ($U_{C_k} \leq 1, \forall\ k \in [1 \dots p]$),
- La période de chaque tâche ne doit pas dépasser sa période maximale ($T_{i_{max}}$), et
- La charge de chaque segment de communication entre deux cœurs ne doit pas dépasser sa capacité maximale ($U_{Com}(C_k, C_l) \leq 1\ , \forall\ k, l \in [1 \dots p]^2$).

Étant donné que l'architecture cible est une architecture multi-cœurs hétérogène et que certaines tâches peuvent être supportées par un ou plusieurs cœurs, nous désignons par **SingleCore task** toute une tâche qui ne peut s'exécuter que sur un et un seul cœur.

Exemple : τ_2 peut s'exécuter que sur C_1 (Figure 4.2).

Avant de décrire les étapes de la stratégie, il est nécessaire de définir d'autres termes qui vont être utilisés par la suite.

Cluster (CL) : Est un groupe de tâches qui échangent de données entre elles.

Exemple : τ_1, τ_2 et τ_3 sont groupées dans CL_1. τ_4 est dans CL_2 (Fig. 4.2).

Common Processor (CP) : Est un cœur qui peut exécuter toutes les tâches d'un *Cluster* tel que son taux d'utilisation soit inférieur ou égal à 1.

Exemple : En regardant la Figure 4.2, nous remarquons que le cœur C_1 peut exécuter toutes les tâches de *Cluster* CL_1. Donc, si son taux d'utilisation (en supportant toutes ces tâches) est inférieur ou égal à 1, alors il est un CP.

CPCL : Est un *Cluster* **CL** possédant un ou plusieurs *Common Processor* **CP**.

Exemple : Si le cœur C_1 est un CP pour CL_1 (Figure 4.2), alors CL_1 devient un **CPCL**.

Data Traffic (DT) : Est la quantité de données transférées (en bits) dans un *Cluster* **CL** en une unité de temps (ut).

Exemple : $DT(CL_1) = ut * (\frac{SM_{1,3} + SM_{1,2}}{T_1})$ et $DT(CL_2) = 0$ (Fig. 4.2).

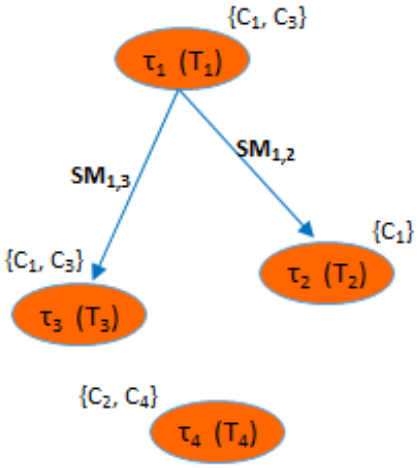

FIGURE 4.1: Exemple d'application sans *Cluster*.

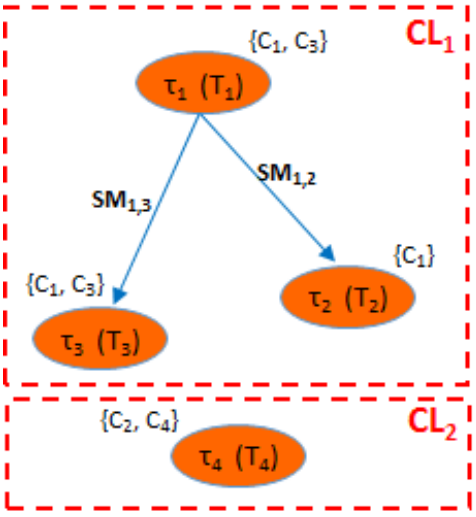

FIGURE 4.2: Exemple d'application avec *Cluster*.

4.4.1.1 Présentation globale des étapes de la stratégie

La structure globale de la stratégie est représentée dans la Figure 4.3. La stratégie est basée sur trois étapes principales présentées comme suit :

Étape 1 : Trouver et placer toutes les *SingleCore tasks*.

Étape 2 : Trouver et placer toutes les tâches communicantes qui ont un (ou des) *Common Processor* CP.

Étape 3 : Placer le reste des tâches (celles qui sont communicantes et qui n'ont pas un *Common Processor* CP).

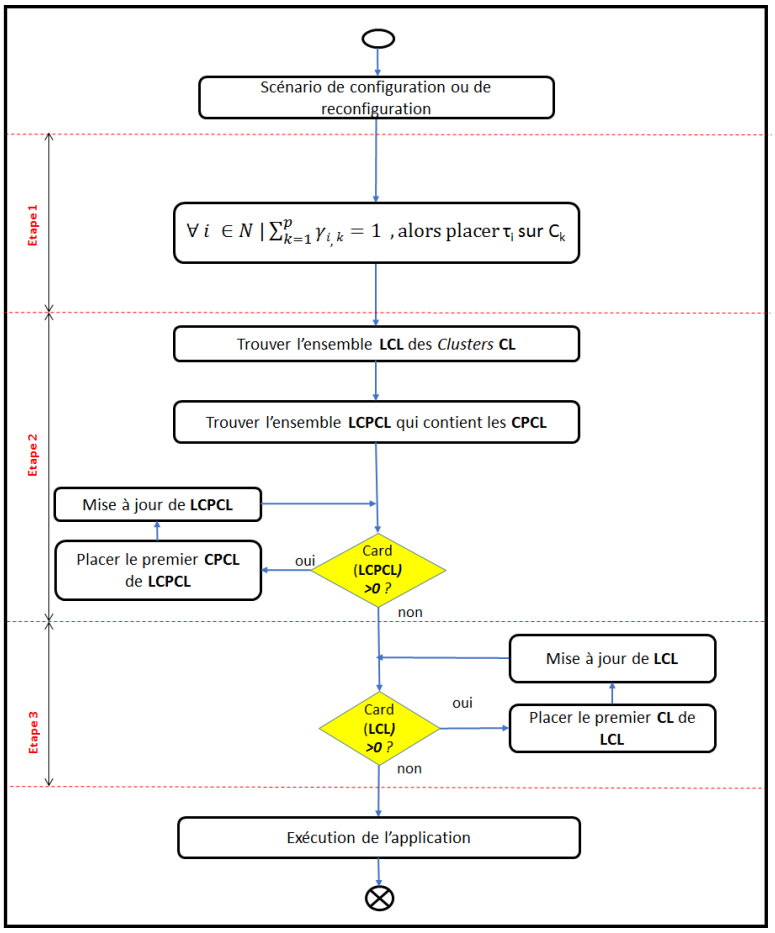

FIGURE 4.3: Aperçu global de l'algorithme de placement de tâches.

4.4.1.2 Présentation détaillée de la stratégie

Dans cette section, nous détaillons chaque étape de la stratégie proposée afin d'expliquer son mode de fonctionnement.

Étant donné que les **SingleCore tasks** ont une position unique, nous supposons qu'elles ont la priorité de placement par rapport à celles qui ont plus de degrés de liberté. En se basant sur cette hypothèse, nous décrivons le principe de la stratégie de placement par les étapes suivantes :

Étape 1 : Placement de toutes les *SingleCore tasks*.

Étape 1.1 : $\forall i \in [1 \dots N] \mid \sum_{k=1}^{p} \gamma_{i,k} = 1$, alors τ_i est placée sur C_k.

Étape 1.2 : Vérification des contraintes.

Étape 2 : Placement des tâches communicantes qui ont un (ou des) *Common Processor* **CP**.

Étape 2.1 : Chaque groupe de tâches communicantes est mis dans un *Cluster* **CL**.

Étape 2.2 : Tous ces *Cluster* seront mis dans une liste **LCL**.

Étape 2.3 : Calculer la quantité de données transférées **DT** pour chaque *Cluster* **CL**.

Étape 2.4 : Trier les *Clusters* **CL** dans l'ordre décroissant selon leurs **DT**.

Étape 2.5 : Chercher les *Common Processor* **CP** pour chaque *Cluster* **CL**.

Étape 2.6 : Chaque *Cluster* **CL** possédant un ou plusieurs *Common Processor* **CP** va être noté par **CPCL**. La liste des **CPCL** est nommée par **LCPCL**.

Étape 2.7 : Mise à jour de la liste **LCL**.

Étape 2.8 : Placer les tâches de chaque **CPCL** sur son *Common Processor* **CP**. S'il existe plus qu'un **CP**, celui le moins chargé sera choisi.

Étape 2.9 : Mise à jours de la liste **LCPCL**.

Étape 2.10 : Retourner à **l'Étape 2.5**.

Étape 3 : Placement des *Clusters* de **LCL** (les groupes des tâches communicantes qui n'ont pas un *Common Processor* **CP**).

Étape 3.1 : En prenant le *Cluster* qui a la plus haute **DT**, placer chaque tâche dans la meilleure position (en terme de coût de communication tout en satisfaisant la contrainte de faisabilité des processeurs).

Étape 3.2 : Mise à jour de la liste **LCL**.

Étape 3.3 : Retourner à **l'Étape 3.1**.

La Figure 4.4 présente en détails les étapes de la stratégie de placement. Par conséquent, la première étape de la stratégie proposée (Figure 4.4) est de placer d'abord les tâches **SingleCore**. Cela peut se faire en les stockant dans un tableau T et en le parcourant case par case afin de placer toutes les **SingleCore task**. Lors du placement des tâches, la stratégie assure la vérification des trois contraintes. S'il y a au moins une contrainte violée, elle propose des modifications au niveau des paramètres de tâches/messages afin de réduire le taux d'utilisation des cœurs et/ou du médium de communication. Si $RTSys$ reste encore non faisable malgré la modification des paramètres, alors la stratégie propose d'enlever certaines tâches jusqu'à l'aboutissement d'un taux d'utilisation qui peut satisfaire toutes les contraintes. Le choix des tâches qui doivent être enlevées se fait selon leurs facteurs d'importances I_i. Dans le cas où il existe plusieurs tâches qui possèdent le même facteur d'importance, la tâche τ_i possédant un taux d'utilisation u_{τ_i} le plus élevé,

devient prioritaire pour être supprimée.

Après avoir placé les **SingleCore tasks**, la stratégie passe à l'étape 2 (Figure 4.4) qui propose de placer les tâches des *Clusters* **CL**. En effet, nous distinguons deux types de *Clusters* : 1) Ceux possédant un (ou des) *Common Processors* notés **CPCL**, c'est-à-dire toutes les tâches d'un *Cluster* peuvent s'exécuter sur un seul cœur et que ce dernier reste faisable. 2) Ceux qui n'ont pas un *Common Processor*. Dans cette étape, nous nous intéressons aux *Clusters* **CPCL**. Une liste **LCPCL** contenant les **CPCL** est construite afin de placer toutes leurs tâches. Nous indiquons que la priorité de placement est donnée aux **CPCL** possédant un trafic de données **DT** le plus élevé. Après chaque placement de **CPCL**, une mise à jour au niveau de **LCPCL** est nécessaire.

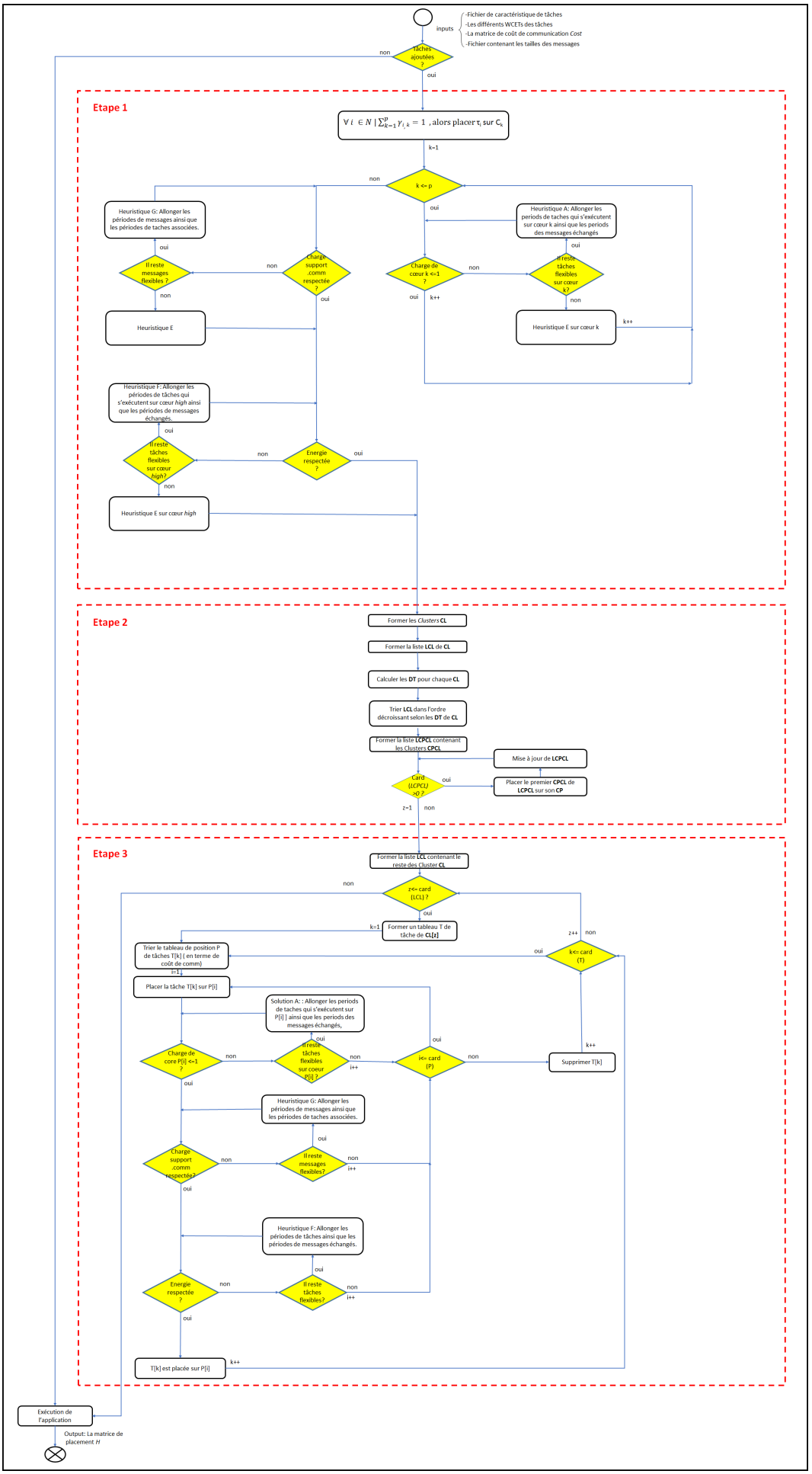

FIGURE 4.4: Stratégie de placement de tâches.

Lorsque toutes les tâches de tous les *Clusters* **CPCL** ont été placées, la stratégie passe à la troisième et dernière étape qui consiste à placer le reste des *Clusters* (ceux qui n'ont pas un *Common Processor*). Ensuite, une liste **LCL** contenant le reste des *Clusters* **CL** est construite. Cette liste est triée dans l'ordre décroissant selon les **DT** de chaque **CL**. Chaque *Cluster* possède un tableau de tâches T qui contient toutes les tâches du *Cluster*. Chaque tâche (représentée par une case du tableau T) pointe sur un autre tableau de position P qui contient les possibilités pour placer cette tâche.

Dans le cas d'une architecture multi-cœurs homogène : il n'y a pas de tâches **SingleCore** puisque toute tâche peut être placée sur tous les cœurs présents dans l'architecture. Ce qui signifie que la première étape n'est pas utile, seules les étapes 2 et 3 sont alors considérées.

4.4.2 Heuristiques utilisées par la stratégie de placement

Lors du placement des tâches, le taux d'utilisation de certains cœurs augmente et cela peut engendrer la violation des contraintes temporelles et/ou énergétiques. De plus, le médium de communication peut être plus chargé s'il y a plus d'échanges, et cela peut engendrer également une violation de contrainte de communication. Par conséquent $RTSys$ risque de devenir non faisable. Nous présentons dans cette section deux nouvelles heuristiques qui s'appliquent lors du placement des tâches afin d'éviter la violation des contraintes de communication et d'énergie. Nous reprenons l'heuristique A, présentée dans le chapitre précédent, pour traiter le cas de la violation de contrainte temps-réel.

Par ailleurs, l'architecture cible dans ce chapitre est une architecture hétérogène, cela veut dire que les WCETs des tâches sont bien définis selon les cœurs (Section 4.2). Par conséquent, une modification du WCET d'une telle tâche peut engendrer sa migration à un autre cœur que nous n'adressons pas dans nos travaux. Pour cela, la solution de modification des WCETs des tâches n'est pas traitée dans ce chapitre.

4.4.2.1 Heuristique F : Modification des périodes des tâches/messages sous contraintes d'énergie totale

Notons que C_{high} ($high \in [1..p]$) le cœur le plus chargé. Nous proposons d'utiliser la stratégie des packs sur les périodes des tâches qui s'exécutent sur C_{high} afin de s'assurer

que la puissance actuelle soit inférieure à la puissance critique.

Nous savons que la contrainte à respecter est la suivante : $P_{Total} \leq P_{Limit}$.

D'après l'Eq. 4.12, on a $P_{Total} = P_{Proc} + P_{Com}$

Donc, $P_{Proc} + P_{Com} \leq P_{Limit}$

i.e.,

$$\sum_{j=1}^{p} P_{C_j} \leq P_{Limit} - P_{Com}$$

$$P_{C_{high}} + \sum_{j=1}^{p-1} P_{C_j} \leq P_{Limit} - P_{Com}$$

Ainsi,

$$P_{C_{high}} \leq P_{Limit} - P_{Com} - \sum_{j=1}^{p-1} P_{C_j}$$

En se référant à l'Eq. 4.9

$$k1 * U_{C_{high}}^2 \leq P_{Limit} - P_{Com} - \sum_{j=1}^{p-1} P_{C_j}$$

Alors,

$$U_{C_{high}} \leq \sqrt{\frac{P_{Limit} - P_{Com} - \sum_{j=1}^{p-1} P_{C_j}}{k1}}$$

Pour simplifier l'écriture des équations, nous posons $\alpha = \sqrt{\frac{P_{Limit} - P_{Com} - \sum_{j=1}^{p-1} P_{C_j}}{k1}}$

Donc,

$$\sum_{i=1}^{N} \left(\frac{W_i}{T_i} * H_{i,high} \right) \leq \alpha$$

Afin de satisfaire la contrainte énergétique et en se basant sur la stratégie des packs, nous regroupons toutes les tâches dans des packs selon leur période. Comme nous l'avons présenté dans le chapitre précédent, le regroupement et le calcul des valeurs des *packs* peuvent se faire sur deux étapes :

Étape 1 : Construction des packs

Similairement au principe de l'heuristique C du chapitre précédent, pour construire les derniers il est nécessaire de chercher la valeur de période convenable pour le premier Pack $Pk^T_{1,high}$ qui minimise le coût de modification des paramètres pour l'ensemble du système. En résolvant le système suivant $Pb7$, nous trouverons $Pk^T_{1,high}$.

$$Pb7 \begin{cases} \text{Minimiser } \sum_{i=1}^{N} (Pk^T_{1,high} - (T_i \ mod \ Pk^T_{1,high})) \ mod \ Pk^T_{1,high} * H_{i,high} \\ t.q. \\ \qquad Pk^T_{1,high} \geq Min(T_i) * H_{i,high}, \ \forall \ i \in [1 \dots N] \end{cases}$$

Une fois que le pack $Pk^T_{1,high}$ est calculé, nous construisons les autres $Packs$ comme suit : $Pack^T_{x,high}, x \geq 1$, contient les tâches qui ont des périodes dans $[(x-1) * Pk^T_{1,high} + 1 \quad ; \quad x * Pk^T_{1,high}]$, $x \in \mathbb{N}^+$.

Il est également ncessaire que chaque nouvelle période ne doit pas dépasser sa période maximale. Cette contrainte est décrite comme suit : $\forall \tau_i \mid \tau_i \in Pack^T_{x,high}, T_{i_{max}} \geq Pk^T_{1,high}$.

Après avoir construit les packs, on vérifie si les nouvelles valeurs satisfont la contrainte énergétique ($\sum_{i=1}^{N} \left(\frac{W_i}{T_i} * H_{i,high} \right) \leq \alpha$) ou pas. Si cette contrainte reste encore violée, on passe à la deuxième étape qui est la recherche des nouvelles valeurs des périodes des tâches tout en se basant sur les packs.

Étape 2 : Recherche des nouvelles périodes des tâches

Puisque les packs sont construits, donc

$$\sum_{\tau_i \in Pack^T_{1,high}} \frac{W_i}{Pk^T_{1,high}} * H_{i,high} + \dots + \sum_{\tau_i \in Pack^T_{x,high}} \frac{W_i}{x.Pk^T_{1,high}} * H_{i,high} \leq \alpha$$

D'où,

$$\frac{1}{Pk^T_{1,high}} * \left(\sum_{\tau_i \in Pack^T_{1,high}} W_i * H_{i,high} + \dots + \sum_{\tau_i \in Pack^T_{x,high}} \frac{W_i}{x} * H_{i,high} \right) \leq \alpha$$

Finalement,

$$Pk^T_{1,high} = \left\lceil \frac{\sum_{\tau_i \in Pack^T_{1,high}} W_i * H_{i,high} + \dots + \sum_{\tau_i \in Pack^T_{x,high}} \frac{W_i}{x} * H_{i,high}}{\alpha} \right\rceil \qquad (4.14)$$

Avec $Pk^T_{1,high}$ la nouvelle période affectée à toutes les tâches du $Pack^T_{1,high}$, les autres packs sont des multiples. Après cette modification, la contrainte énergétique peut être satisfaite. Par ailleurs, il faut vérifier la contrainte suivante afin de ne pas dépasser les périodes maximales des tâches : $\forall \tau_i \mid \tau_i \in Pack^T_{x,high}, T_{i_{max}} \geq Pk^T_{x,high}$

Comme nous l'avons indiqué au début du chapitre, chaque modification au niveau des périodes des tâches implique implicitement une modification sur les périodes des messages associés.

4.4.2.2 Heuristique G : Modification des périodes des messages/tâches sous contrainte de communication

Si $U_{Com}(C_k, C_l) > 1$ $(\forall k, l \in [1 \dots p]^2)$, alors la contrainte de communication est violée. En utilisant la stratégie des packs, nous pouvons résoudre ce problème en allongeant les périodes des messages qui circulent entre les cœurs C_k et C_l. Similairement aux précédents travaux, la solution est basée sur deux étapes :

Étape 1 : Construction des packs

En utilisant le même principe de la construction des packs dans le chapitre précédent, nous construisons les packs par la résolution du problème suivant :

$$Pb8 \begin{cases} \text{Minimiser } \sum_{i=1}^{N}\sum_{j=1}^{N}(Pk^{TM}_{1,k,l} - (TM_{i,j} \ mod \ Pk^{TM}_{1,k,l})) \ mod \ Pk^{TM}_{1,k,l} * H_{i,k} * H_{i,l} \\ t.q. \\ \qquad Pk^{TM}_{1,k,l} \geq Min(TM_{i,j}), \ \forall \ i,j \in [1 \dots N] \end{cases}$$

Une fois $Pk^{TM}_{1,k,l}$ calculée, les packs des messages sont organisés comme suit : $Pack^{TM}_{x,k,l}$ contient les messages qui ont des périodes dans $[(x-1) * Pk^{TM}_{x,k,l} + 1 \quad ; \quad x * Pk^{TM}_{1,k,l}]$, $x \in \mathbb{N}^+$.
Par ailleurs, il faut vérifier que chaque période de message ne dépasse pas la période maximale de sa tâche émettrice : $\forall M_{i,j} \mid M_{i,j} \in Pack^{TM}_{x,k,l}, T_{i_{max}} \geq Pk^{TM}_{x,k,l}$.

Après avoir construit les packs et allongé les périodes des messages, on vérifie si la contrainte de communication devient respectée ou pas. Si elle ne l'est pas, on passe l'Étape 2 qui est la recherche des nouvelles valeurs des périodes de messages.

Étape 2 : Recherche des nouvelles périodes des messages

Pour satisfaire la contrainte de communication, il est nécessaire de vérifier :

$$\sum_{i=1}^{N}\sum_{j=1}^{N}\left(\frac{WM_{i,j}}{TM_{i,j}} * H_{i,k} * H_{j,l}\right) \leq 1$$

En regroupant les messages dans des packs, nous obtenons :

$$\sum_{M_{i,j}\in Pack_{1,k,l}^{TM}} \frac{WM_{i,j}}{Pk_{1,k,l}^{TM}} * H_{i,k} * H_{j,l} + ... + \sum_{M_{i,j}\in Pack_{x,k,l}^{TM}} \frac{WM_{i,j}}{x.Pk_{1,k,l}^{TM}} * H_{i,k} * H_{j,l} \leq 1$$

Ainsi,

$$\frac{1}{Pk_{1,k,l}^{TM}} * \left(\sum_{M_{i,j}\in Pack_{1,k,l}^{TM}} WM_{i,j} * H_{i,k} * H_{j,l} + ... + \sum_{M_{i,j}\in Pack_{x,k,l}^{TM}} \frac{WM_{i,j}}{x} * H_{i,k} * H_{j,l}\right) \leq 1$$

Puisque les périodes sont des entiers, alors,

$$Pk_{1,k,l}^{TM} = \left\lceil \sum_{M_{i,j}\in Pack_{1,k,l}^{TM}} WM_{i,j} * H_{i,k} * H_{j,l} + ... + \sum_{M_{i,j}\in Pack_{x,k,l}^{TM}} \frac{WM_{i,j}}{x} * H_{i,k} * H_{j,l} \right\rceil \tag{4.15}$$

Avec $Pk_{1,k,l}^{TM}$ la nouvelle période affectée à toutes les messages du $Pack_{1,k,l}^{TM}$, les autres packs sont des multiples. Après l'allongement des périodes des messages et implicitement celles des tâches produisant les messages, la contrainte de communication peut être vérifiée. Si elle ne l'est pas, il faut passer à la suppression de certaines tâches.

4.5 Implémentation et discussion

Cette section est consacrée à l'implémentation et l'évaluation de la stratégie proposée ainsi que l'évaluation de sa performance. Nous introduisons dans la première sous section une étude de cas pour décrire en détail l'enchaînement de la stratégie proposée. Dans la seconde sous section, nous nous intéressons à la comparaison de notre stratégie avec d'autres algorithmes de placement de tâches cités dans l'état de l'art.

4.5.1 Etude de cas

Cette sous section présente une étude de cas qui permet de mettre en place la stratégie proposée. Avant d'appliquer les étapes de la stratégie, nous commençons d'abord par présenter les paramètres du problème à résoudre.

4.5.1.1 Paramètres d'entrée du problème à résoudre

Nous supposons que $RTSys$ est composé de l'ensemble de tâches : $\Pi = \{\tau_1, \tau_2, ..., \tau_{10}\}$ et d'un ensemble de trois cœurs hétérogènes $\Gamma=\{C_1, C_2, C_3\}$ inter-connectés par une P2P [106] (Figure. 4.5).

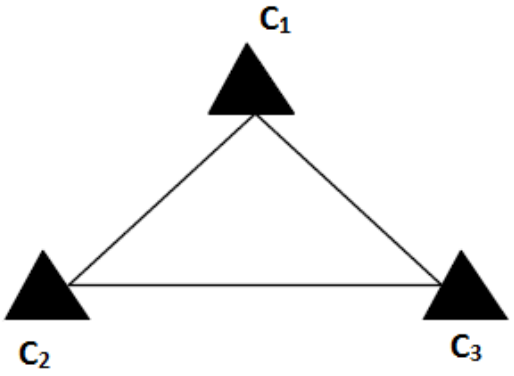

FIGURE 4.5: Liaison P2P de trois cœurs.

L'objectif est d'appliquer la stratégie proposée pour placer les tâches sur les différents cœurs tout en gardant la faisabilité du $RTSys$. Après le placement des tâches, il est aussi important d'avoir un coût de communication minimal.

Dans un premier temps, il est nécessaire de donner la matrice des coût de communication $Cost$ qui correspond à la topologie P2P.

$$Cost = \begin{array}{c} \begin{array}{ccc} C_1 & C_2 & C_3 \\ \downarrow & \downarrow & \downarrow \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} \\ \leftarrow C_1 \\ \leftarrow C_2 \\ \leftarrow C_3 \end{array}$$

La communication entre les tâches est illustrée dans la Figure 4.6.

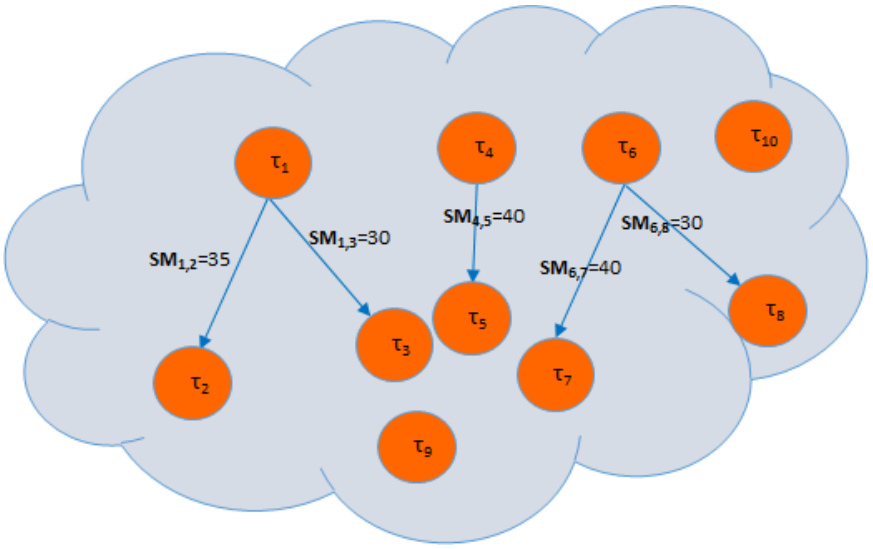

FIGURE 4.6: Graphe de la communication inter-tâches.

La matrice γ qui représente les différentes possibilités de placement de chaque tâche est représentée comme suit :

$$\gamma = \begin{array}{c} \begin{array}{ccc} C_1 & C_2 & C_3 \\ \downarrow & \downarrow & \downarrow \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \end{array} \begin{array}{l} \leftarrow \tau_1 \\ \leftarrow \tau_2 \\ \leftarrow \tau_3 \\ \leftarrow \tau_4 \\ \leftarrow \tau_5 \\ \leftarrow \tau_6 \\ \leftarrow \tau_7 \\ \leftarrow \tau_8 \\ \leftarrow \tau_9 \\ \leftarrow \tau_{10} \end{array}$$

Les caractéristiques des tâches sont illustrées dans les Tableaux 4.1 et 4.2. Si une tâche ne peut pas être exécutée par un cœur donné, alors on note la valeur de son WCET par "NaN".

Les caractéristiques des messages échangés entres les tâches sont illustrées dans le Tableau 4.3.

TABLE 4.1: Caractéristiques des tâches initiales.

Tâche	T_i	$T_{i_{max}}$	I_i
τ_1	5	6	1
τ_2	6	8	2
τ_3	10	10	1
τ_4	4	6	3
τ_5	12	22	8
τ_6	5	16	10
τ_7	8	21	12
τ_8	8	32	14
τ_9	6	14	7
τ_{10}	7	12	3

TABLE 4.2: WCET W_i des tâches.

WCET	C_1	C_2	C_3
W_1	1	3	NaN
W_2	1	NaN	2
W_3	2	NaN	NaN
W_4	1	2	NaN
W_5	2	NaN	NaN
W_6	NaN	2	NaN
W_7	NaN	NaN	3
W_8	NaN	2	1
W_9	NaN	NaN	2
W_{10}	NaN	2	1

TABLE 4.3: Caractéristiques des messages initiaux échangés.

Message	$WM_{i,j}$	$TM_{i,j}$	$SM_{i,j}$
$M_{1,2}$	1	5	35
$M_{1,3}$	1	5	30
$M_{4,5}$	1	4	40
$M_{6,7}$	2	5	40
$M_{6,8}$	2	5	30

4.5.1.2 Application des étapes de la stratégie

Étape 1 : Placement de tous les **SingleCore tasks**.

Dans cette étape, nous chercherons les tâches possédant une seule possibilité de placement. Selon la matrice donnée γ, les tâches τ_3, τ_5, τ_6, τ_7 et τ_9 doivent être placées sur leurs cœurs appropriés comme illustré dans le Tableau 4.4.

TABLE 4.4: Placement de SingleCore task.

SingleCore tasks	cœur
τ_3	C_1
τ_5	C_1
τ_6	C_2
τ_7	C_3
τ_9	C_3

Après avoir placé les **SingleCore tasks**, nous remarquons que le message $M_{6,7}$ est ajouté sur le segment entre les cœurs C_2 et C_3. L'utilisation des processeurs ainsi que l'utilisation

des segments de communication sont illustrées dans le Tableau 4.5.

TABLE 4.5: Utilisation des processeurs après l'exécution de l'Étape 1.

	C_1	C_2	C_3	$Comm(C_1, C_2)$	$Comm(C_1, C_3)$	$Comm(C_2, C_3)$
U	0.36	0.4	0.708	0	0	0.4

Étape 2 : Placement des tâches communicantes qui ont un (ou des) *Common Processors* **CP**.

Dans cette étape, nous formons les *Clusters* et cherchons s'il y a des *Clusters* possédant des *Common Processors* tout en les triant en ordre décroissant selon leurs **DT**. Le calcul de **DT** peut être réalisé en utilisant la formule présentée dans la section 4.4.1. Le Tableau 4.6 illustre les différents *Clusters* du graphe de tâches de la Figure 4.6.

TABLE 4.6: Classification des tâches en *Clusters*.

Clusters	Tâches	DT	CP
$CPCL_1$	τ_1, τ_2, τ_3	$\frac{35+30}{5}$=13	C_1
$CPCL_2$	τ_4, τ_5	$\frac{40}{4}$=10	C_1
$CPCL_3$	τ_9	0	C_3
$CPCL_4$	τ_{10}	0	C_2, C_3
CL_1	τ_6, τ_7, τ_8	$\frac{40+30}{5}$=14	-

Puisque les tâches τ_3, τ_5, τ_6, τ_7 et τ_9 ont été déjà placées dans l'étape 1, les *Clusters* sont de nouveau illustrés dans le Tableau 4.7.

TABLE 4.7: Mise à jour des *Clusters*.

Clusters	Tâches	DT	CP
$CPCL_1$	τ_1, τ_2	13	C_1
$CPCL_2$	τ_4	10	C_1
$CPCL_3$	τ_{10}	0	C_2, C_3
CL_1	τ_8	14	-

Selon le Tableau 4.6, il s'agit de quatre *Clusters* dont trois sont des **CPCL**. Nous commençons donc par placer les tâches du *Cluster* $CPCL_1$ sur leur *Common Processor*, nous plaçons donc τ_1 et τ_2 sur C_1. L'utilisation du cœur C_1 après l'ajout des tâches augmente pour atteindre 0.732 et sa charge est donc encore acceptable. Il faut noter qu'avant de passer au placement du *Cluster* suivant, il faut mettre à jour les taux d'utilisation des cœurs ainsi que la liste des **CPCL**. De nouveaux taux d'utilisation des cœurs ainsi que des **DT** sont respectivement illustrées dans les Tableaux 4.8 et 4.9.

TABLE 4.8: Mise à jour des taux d'utilisation des cœurs.

	C_1	C_2	C_3	$Comm(C_1, C_2)$	$Comm(C_1, C_3)$	$Comm(C_2, C_3)$
U	0.732	0.4	0.708	0	0	0.4

TABLE 4.9: Nouvelle mise à jour des *Clusters*.

Clusters	Tâches	DT	CP
$CPCL_2$	τ_4	10	C_1
$CPCL_3$	τ_{10}	0	C_2, C_3
CL_1	τ_8	14	-

Nous passons maintenant au placement de τ_4 du deuxième **CPCL** qui va être placée bien évidement sur C_1. Ainsi, la nouvelle valeur de U_{C_1} est égale à 0.982.

Selon la matrice γ, τ_{10} du troisième *Cluster* possède deux possibilités de placement soit sur C_2 soit sur C_3. En équilibrant les charges des cœurs, τ_{10} doit être placée sur C_2 et U_{C_2} devient 0.685. D'où, la faisabilité du $RTSys$ est toujours vérifiée.

Puisque toutes les tâches des **CPCL** sont bien placées, nous présentons dans le Tableau 4.10 les nouveaux taux d'utilisation des cœurs ainsi que les segments de communication entre cœurs.

TABLE 4.10: Utilisation du cœur après l'exécution de l'Étape 2.

	C_1	C_2	C_3	$Comm(C_1, C_2)$	$Comm(C_1, C_3)$	$Comm(C_2, C_3)$
U	0.982	0.685	0.708	0	0	0.4

Nous passons maintenant au placement des *Clusters* qui n'ont pas un *Common Processor*.

Étape 3 : Placement des tâches communicantes qui n'ont pas un *Common Processor* **CP**.

Dans cette étape, il reste à placer un seul *Cluster* qui contient la tâche τ_8. Cette tâche τ_8 peut être placée sur l'un des deux cœurs C_2 et C_3. Dans ce cas, la stratégie propose de choisir le placement qui minimise le coût de communication. Vu que τ_8 possède des communications avec τ_6, alors ce serait mieux de la mettre sur le même cœur qui exécute τ_6 afin d'éliminer le coût de communication entre elles. Nous choisissons donc de placer τ_8 sur C_2. Après l'ajout de cette tâche, l'utilisation du C_2 devient 0.935 et reste faisable.

Après avoir placé toutes les tâches, la stratégie fournit la matrice de *mapping* comme *Output*. Cette matrice est illustrée comme suit :

$$H = \begin{array}{c} \begin{array}{ccc} C_1 & C_2 & C_3 \\ \downarrow & \downarrow & \downarrow \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} \leftarrow \tau_1 \\ \leftarrow \tau_2 \\ \leftarrow \tau_3 \\ \leftarrow \tau_4 \\ \leftarrow \tau_5 \\ \leftarrow \tau_6 \\ \leftarrow \tau_7 \\ \leftarrow \tau_8 \\ \leftarrow \tau_9 \\ \leftarrow \tau_{10} \end{array}$$

Le coût total de communication $TotalCost$ est 40 unités. Le tableau ci-dessous (4.11) est un tableau récapitulatif des utilisations des cœurs/segments de communication.

TABLE 4.11: Les dernières valeurs d'utilisation des cœurs après le *mapping*.

	C_1	C_2	C_3	$Comm(C_1, C_2)$	$Comm(C_1, C_3)$	$Comm(C_2, C_3)$
U	0.982	0.935	0.708	0	0	0.4

4.5.1.3 Application d'un premier scénario de reconfiguration

Nous supposons qu'à l'instant t_i, $RTSys$ subit un scénario de reconfiguration en ajoutant deux autres tâches τ_{11} et τ_{12}. Figure 4.7 représente la nouvelle communication inter-tâches.

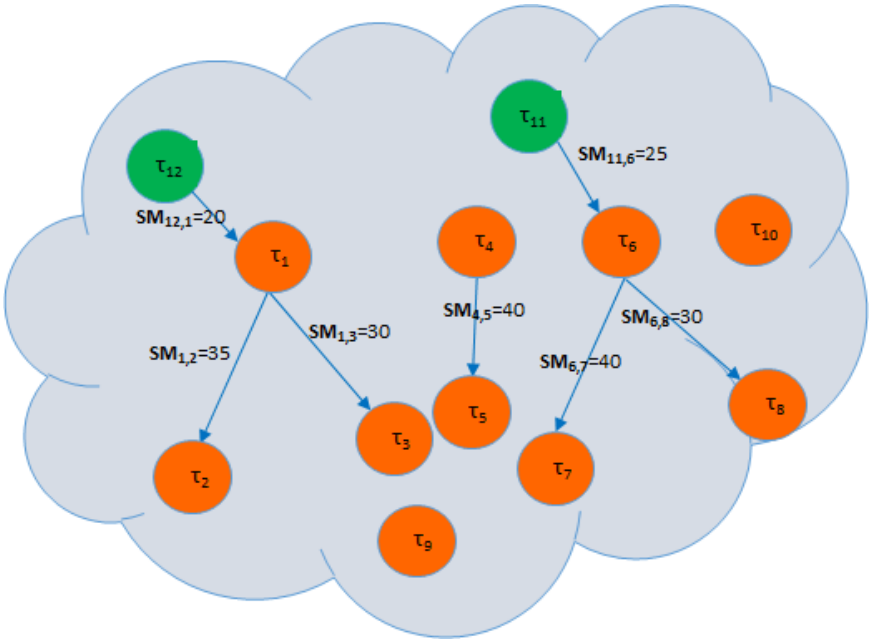

FIGURE 4.7: Graphe de la communication inter-tâches après l'ajout d'autres tâches.

$$\gamma = \begin{array}{c} \begin{array}{ccc} C_1 & C_2 & C_3 \\ \downarrow & \downarrow & \downarrow \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{l} \\ \leftarrow \tau_{11} \\ \leftarrow \tau_{12} \end{array}$$

Les Tableaux 4.12 et 4.13 illustrent les caractéristiques des nouvelles tâches

TABLE 4.12: Tâches ajoutées après le scénario de reconfiguration.

Tâche	T_i	$T_{i_{max}}$	I_i
τ_{11}	12	20	8
τ_{12}	4	15	12

TABLE 4.13: WCET W_i de chaque nouvelles tâches.

WCET	C_1	C_2	C_3
W_{11}	NaN	2	NaN
W_{12}	1	NaN	1

Tableau 4.14 représente les caractéristiques des nouveaux messages.

TABLE 4.14: Caractéristiques des nouveaux messages.

Message	$WM_{i,j}$	$TM_{i,j}$	$SM_{i,j}$
$M_{11,6}$	2	12	25
$M_{12,1}$	1	4	20

Étape 1 : Placement de tous les **SingleCore tasks**.

Selon la matrice γ qui représente les possibilités de placement de τ_{11} et τ_{12}, nous remarquons que la tâche τ_{11} est une **SingleCore task** qui a la priorité de placement sur son unique cœur C_2.

Après l'ajout de τ_{11}, le cœur C_2 viole sa contrainte temporelle puisque son taux d'utilisation dépasse la valeur maximale 1 ($U_{C_2} = 0.935 + \frac{2}{12} = 1.101$) et par conséquent $RTSys$ devient non faisable. Pour résoudre ce problème, nous retournons au logigramme de la stratégie dans la Figure 4.4. Dans ce cas de violation de contraintes temporelles, notre stratégie propose d'appliquer l'heuristique A qui allonge les périodes des tâches pour baisser l'utilisation du cœur C_2 afin de satisfaire cette contrainte et réobtenir la faisabilité du $RTSys$.

Rappelons que le cœur C_2 exécute les quatre tâches τ_6, τ_8, τ_{10} et τ_{11} avec un taux d'utilisation égal à 1.101. En se basant sur le principe de l'heuristique A présenté dans la sous section 3.4.1.1, nous cherchons d'abord la nouvelle période du premier pack en utilisant l'Eq. 3.7. Les autres packs sont des multiples du premier pack. Après avoir appliqué l'heuristique choisie, l'utilisation du cœur C_2 a été baissée à 0.933 et nous présentons les nouvelles périodes des tâches dans le Tableau 4.15.

TABLE 4.15: Les nouvelles périodes des tâches qui s'exécutent sur C_2.

Tâche	Ancien T_i	Nouvelle T_i	$T_{i_{max}}$
τ_6	5	5	16
τ_8	8	10	32
τ_{10}	7	10	12
τ_{11}	12	15	20

Chaque modification d'un paramètre de tâche τ_i a un coût noté $CoûtModif_{\tau_i}$ qui est égal à la différence entre la nouvelle valeur et la valeur initiale du paramètre modifié. Le coût total de modification est la somme de tous les $CoûtModif_{\tau_i}$. Dans cet exemple, on a : $CoûtModif_{\tau_6}$=5-5=0, $CoûtModif_{\tau_8}$= 10-8=2, $CoûtModif_{\tau_{10}}$= 10-7= 3, et $CoûtModif_{\tau_{11}}$= 15-12= 3. D'où, le coût total de modification des périodes $CoûtTotalModif$ est égale à 0+2+3+3=8.

Étape 2 : Placement des tâches communicantes qui ont un (ou des) *Common Processors* **CP**.

La dernière tâche qui doit être placée c'est la tâche τ_{12}. Elle a deux possibilités de placement soit sur C_1 soit C_3. L'objectif de cette deuxième étape est de minimiser le coût de communication entre cœurs en plaçant les tâches sur un $CommunProcessor$ CP. Vu que τ_{12} est en communication avec τ_1, nous essayons de la placer sur le cœur

exécutant τ_1. Selon la matrice de *mapping* H, nous remarquons que τ_1 a été placée sur C_1. Par conséquent, nous proposons de placer τ_{12} aussi sur C_1. Néanmoins, le taux d'utilisation devient $0.982 + \frac{1}{4} = 1.232$ ce qui n'est pas acceptable puisque la contrainte temporelle est violée. L'idée est donc d'appliquer l'heuristique A qui permet d'allonger les périodes des tâches. Malheureusement, après l'application de l'heuristique A, nous ne parvenons pas à abaisser suffisamment le taux d'utilisation du cœur à une valeur inférieure à 1 car les $T_{i_{max}}$ des tâches qui s'exécutent sur C_1 sont très proches de leurs périodes nominales T_i.

Étape 3 : Placement des tâches communicantes qui n'ont pas un *Common Processor* **CP**.

Cette étape propose de placer τ_{12} sur le second cœur possible qui est C_3. L'utilisation du cœur C_3 devient donc $0.708 + \frac{1}{4} = 0.958$. Notons que toutes les tâches sont placées et que $RTSys$ reste faisable.
La nouvelle matrice de *mapping* H est le suivant :

$$H = \begin{array}{c} \begin{array}{ccc} C_1 & C_2 & C_3 \\ \downarrow & \downarrow & \downarrow \end{array} \\ \left(\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right) \end{array} \begin{array}{l} \\ \leftarrow \tau_1 \\ \leftarrow \tau_2 \\ \leftarrow \tau_3 \\ \leftarrow \tau_4 \\ \leftarrow \tau_5 \\ \leftarrow \tau_6 \\ \leftarrow \tau_7 \\ \leftarrow \tau_8 \\ \leftarrow \tau_9 \\ \leftarrow \tau_{10} \\ \leftarrow \tau_{11} \\ \leftarrow \tau_{12} \end{array}$$

Le coût de communication total $TotalCost$ est 40+20=60 unités. L'utilisation des cœurs ainsi que l'utilisation des segments de communication sont illustrées dans le Tableau 4.16.

TABLE 4.16: Utilisation du cœur après le scénario de reconfiguration.

	C_1	C_2	C_3	$Comm(C_1, C_2)$	$Comm(C_1, C_3)$	$Comm(C_2, C_3)$
U	0.982	0.933	0.958	0	0.25	0.4

4.5.1.4 Application d'un second scénario de reconfiguration

Nous supposons qu'à l'instant t_k $(k > i)$, *RTSys* subit un deuxième scénario de reconfiguration en ajoutant une autre tâches τ_{13} qui communique avec la tâche τ_{12}. Figure 4.8 représente la nouvelle communication inter-tâches.

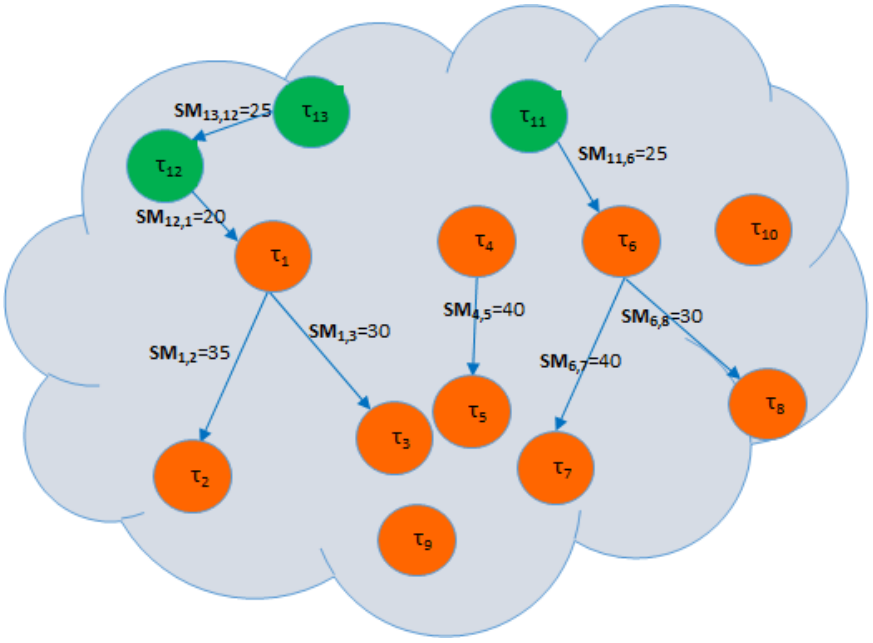

FIGURE 4.8: Graphe de la communication inter-tâches après l'ajout de τ_{13}.

$$\gamma = \begin{matrix} C_1 & C_2 & C_3 \\ \downarrow & \downarrow & \downarrow \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 \end{pmatrix} \leftarrow \tau_{13}$$

Les Tableaux 4.17 et 4.18 illustrent les caractéristiques de la nouvelle tâche. Concernant les caractéristiques du message $M_{13,12}$, ils sont illustrés par le Tableau 4.19. Étant donné

TABLE 4.17: Tâche ajoutée après le deuxième scénario de reconfiguration.

Tâche	T_i	$T_{i_{max}}$	I_i
τ_{13}	5	6	15

que la tâche τ_{13} n'est pas une **SingleCore task**, alors l'Étape 1 ne sera pas appliquée et on passe directement à l'Étape 2.

TABLE 4.18: WCET W_i de chaque nouvelles tâches.

WCET	C_1	C_2	C_3
W_{13}	NaN	1	2

TABLE 4.19: Caractéristiques des nouveaux messages.

Message	$WM_{i,j}$	$TM_{i,j}$	$SM_{i,j}$
$M_{13,12}$	1	5	25

Étape 2 : Placement des tâches communicantes qui ont un (ou des) *Common Processors* **CP**.

Vu que τ_{13} est en communication avec τ_{12}, nous essayons de la placer sur le cœur exécutant τ_{12}. Selon la matrice de *mapping* H, nous remarquons que τ_{12} a été placée sur C_3. En essayant de placer τ_{13} sur C_3, le taux d'utilisation U_{C_3} devient $0.958 + \frac{2}{5} = 1.358$ ce qui n'est pas acceptable puisque la contrainte temporelle est violée. Même avec la l'application de l'heuristique A, nous ne parvenons pas à abaisser suffisamment le taux d'utilisation du cœur à une valeur inférieur à 1, car $T_{13_{max}}$ est très proche de T_{13}.

Étape 3 : Placement des tâches communicantes qui n'ont pas un *Common Processor* **CP**.

Cette étape propose de placer τ_{13} sur le second cœur possible qui est C_2. L'utilisation du cœur C_2 devient donc $0.933 + \frac{1}{5} = 1.133$ ce qui viole la contrainte temporelle. Rappelons que la tâche τ_{13} possède une période maximale qui est égale 6, ce qui ne permet pas de la mettre dans le deuxième pack (pack contant les tâches de périodes 10) et le système reste non faisable. Pour résoudre ce problème, nous retournons au logigramme de la stratégie (Figure 4.4). Dans ce cas de violation de contraintes, notre stratégie supprime la tâche ajoutée puisqu'elle n'est pas importance (elle a la plus haute I_i qui est égale à 15). Donc la tâche τ_{13} sera supprimée et nous gardons la même ancienne matrice de *mapping H*.

4.5.2 Simulations

Pour montrer l'efficacité de la stratégie proposée, nous avons réalisé des comparaisons avec les deux algorithmes de placement de Bhardwaj et al. [22] et Govil et al. [21]. Cette section permet donc de situer la stratégie proposée par rapport aux algorithmes mentionnés ci-dessus. Afin d'effectuer une évaluation plus convaincante, nous proposons d'implémenter les trois approches en utilisant les mêmes études de cas présentées dans

[22]. L'objectif est de placer les tâches sur les différents cœurs et de calculer le coût total de communication de chaque approche. Le support de communication *1-D Mesh network* [107] est utilisé pour connecter les cœurs. La matrice qui présente le coût de communication entre les cœurs selon ce type de réseau est la suivante :

$$Cost = \begin{array}{c} \begin{array}{cccc} C_1 & C_2 & \cdots & C_p \\ \downarrow & \downarrow & \cdots & \downarrow \end{array} \\ \begin{pmatrix} 0 & 1 & \cdots & p-1 \\ 1 & 0 & \cdots & p-2 \\ \vdots & \ddots & \ddots & \vdots \\ p-1 & \cdots & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} \\ \leftarrow C_1 \\ \leftarrow C_2 \\ \vdots \\ \leftarrow C_p \end{array}$$

Les Tableaux 4.20 et 4.21 représentent le coût de communication pour chaque algorithme. Les Figures 4.9 et 4.10 présentent des courbes comparatives qui comparent le coût de communication de chaque approche.

4.5.2.1 Comparaison des algorithmes en augmentant le nombre de tâches

Dans un premier temps, la comparaison est effectuée en fixant le nombre de cœurs avec une variation du nombre de tâches (Tableau 4.20). Selon la Figure 4.9, la courbe orange représente le coût de communication en appliquant l'algorithme proposé par Govil et al. [21] et celle en bleue indique le coût de communication après l'application de l'algorithme proposé par Bhardwaj et al. [22]. Le coût de communication, lorsque nous appliquons notre stratégie est représenté par la courbe grise.

TABLE 4.20: Coûts de communication des trois algorithmes en augmentant le nombre de tâches.

Nombre de tâches	Algo. Bhardwaj et al. [22]	Algo. Govil et al. [21]	Stratégie proposée
6	101	106	90
7	135	140	112
8	169	188	135
9	224	243	201
10	251	259	222
11	319	333	290
12	365	378	345
13	401	435	375
14	465	480	450
15	502	518	472
16	522	533	500
17	545	565	523
18	576	588	552
19	591	617	550
20	620	652	592
Total	5786	6035	5409

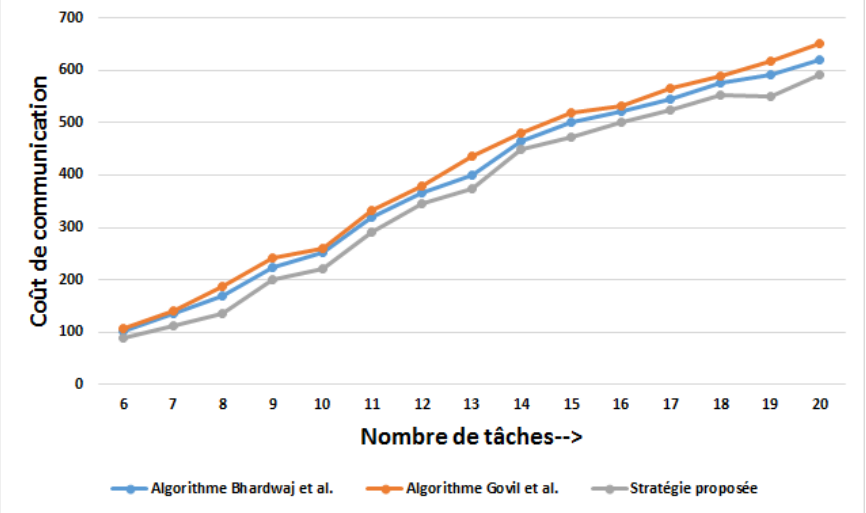

FIGURE 4.9: Coût total de communication en augmentant le nombre de tâches, avec un nombre de cœurs fixé à 4.

Les résultats expérimentaux sur notre stratégie montrent qu'elle offre 11% ($\frac{6035-5409}{6035} * 100$) de gain de coût de communication par rapport à l'algorithme proposé par Govil et al. [21]. Elle offre 7% ($\frac{5786-5409}{5786} * 100$) de gain par rapport à l'algorithme proposé par Bhardwaj et al. [22]. L'explication tient au fait que l'algorithme de Bhardwaj et al. [22] est une extension améliorée de celui proposé par Govil et al. [21].

4.5.2.2 Comparaison des algorithmes en augmentant le nombre de cœurs

Nous augmentons, dans un second temps, le nombre de cœurs en fixant le nombre de tâches (Tableau 4.21 et Figure 4.10). Les simulations montrent que notre stratégie est meilleure par rapport à celles proposées dans [22] et [21]. Ceci est expliqué par l'efficacité de la stratégie puisqu'elle offre plusieurs heuristiques ce qui donne plus de flexibilité au niveau du placement.

TABLE 4.21: Coûts de communication des trois algorithmes en augmentant le nombre de cœurs.

Nombre de cœurs	Algo. Bhardwaj et al. [22]	Algo. Govil et al. [21]	Stratégie proposée
4	620	645	608
5	640	672	630
6	640	669	625
7	650	678	638
8	649	675	636
9	647	672	635
10	655	682	646
11	666	688	658
12	665	685	659
13	660	685	655
14	658	678	650
15	655	675	645
Total	7805	8104	7685

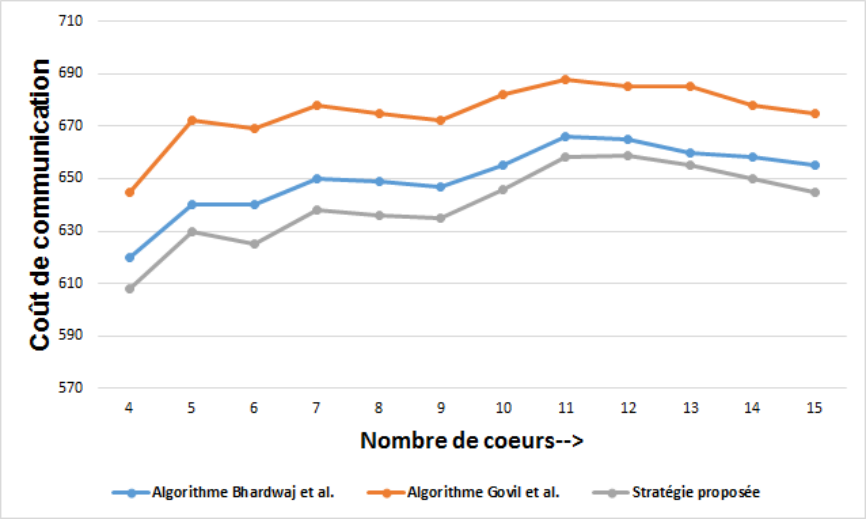

FIGURE 4.10: Coût total de communication en augmentant le nombre de cœurs, avec un nombre de tâches fixé à 20.

En mesurant le gain obtenu en terme de coût de communication, nous constatons que notre stratégie offre un peu plus de 5% ($\frac{8104-7685}{8104} * 100$) de gain par rapport à l'algorithme [21] et 2% ($\frac{7805-7685}{7805} * 100$) de gain par rapport à celui proposé par Bhardwaj et al. [22].

4.5.2.3 Comparaison des algorithmes en termes de taux de rejet de tâches

L'application des trois algorithmes peut conduire à des rejets de tâches, pour des raisons de violation soit i) des contraintes de charge des cœurs, ii) des contraintes de charge des liens (virtuels) de communication et iii) des contraintes d'énergie. Pour évaluer les performances de la stratégie que nous proposons, nous évaluons donc ce critère pour les trois algorithmes précités.

Nous exploitons le module $Task - generator$ présenté dans le chapitre précédant pour générer aléatoirement une application de 800 tâches. Nous indiquons ici qu'avant de lancer la génération des tâches, nous configurons le simulateur comme suit (Tableau 4.22) :

TABLE 4.22: Distribution uniforme des paramètres de tâches.

Paramètre de tâche	Min	Max
WCET W	1	10
Période T	200	1000
Période maximale T_{max}	T	2*T

Après avoir généré aléatoirement une application de 800 tâches, nous appliquons les trois algorithmes afin d'évaluer le taux de rejet des tâches pour chacun des algorithmes. La Figure 4.11 présente le taux de rejet en fonction du nombre de cœurs. En utilisant les mêmes couleurs des courbes, nous pouvons conclure que la stratégie proposée rejette moins de tâches lors du placement. En d'autres termes, notre stratégie offre 62% de gain au niveau taux de rejet de tâches par rapport à l'algorithme [21] et 59% de gain par rapport à celui présenté dans [22]. L'explication tient au fait que notre stratégie propose de modifier les paramètres de tâches avant de passer à la dernière étape qui est le rejet de tâches.

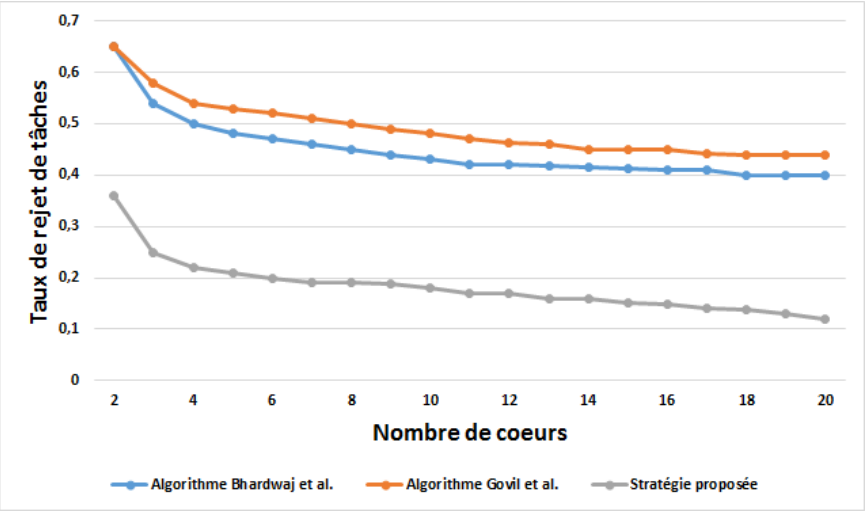

FIGURE 4.11: Taux de rejet de tâches en fonction du nombre des cœurs.

4.6 Conclusion

Dans ce chapitre, nous nous sommes intéressés au placement des tâches sur une architecture multi-cœurs hétérogènes. Nous avons proposé une stratégie permettant de placer les tâches afin de minimiser la consommation énergétique du système. Notre stratégie propose un placement de tâches qui réduit le coût de communication entre cœurs en plaçant celles qui communiquent fréquemment sur le même cœur lorsque cela est possible. Elle propose également de réduire la distance entre ces tâches si le placement sur le même cœur n'est pas possible. Cette stratégie engendre un gain appréciable vis-à-vis à des critères considérés importants dans les systèmes embarqués, notamment le coût de communication et le taux de rejet de tâches qui est de 59% par rapport à l'approche proposée par [22]. Cette contribution est originale puisque, à notre connaissance, aucune recherche n'a traité le cas de placement de tâches pour un système reconfigurable par l'ajout des tâches partageant des messages, tout en respectant les contraintes temporelles et énergétiques.

Chapitre 5

Conclusion générale et perspectives

5.1 Contexte et problématique

La conception de systèmes temps-réel embarqués se développe de plus en plus avec l'intégration croissante de fonctionnalités critiques dans les processus industriels et ces systèmes doivent relever de nombreux défis, en particulier celui de la minimisation de consommation d'énergie. Dans le cas des systèmes des réseaux de capteurs sans fil qui se rechargent périodiquement grâce à une source d'énergie renouvelable, ils doivent fonctionner en continue malgré dans des périodes d'absence de source d'énergie (le soleil dans ce cas particulier de capteurs alimentés par des panneaux solaires). Vu qu'il y a des perturbations qui peuvent avoir lieu suite à des événements externes (conditions climatiques défavorables) ou internes (surcharge d'activité), le système doit être agile et adapter sa partie logicielle en termes de nouvelles et anciennes tâches logicielles sans arrêter son exécution en assurant, autant que possible, un fonctionnement continu. Ceci est un nouveau challenge dans l'industrie pour les prochaines générations de systèmes embarqués. Ces systèmes embarqués doivent s'adapter à leur environnement par l'ajout et/ou de suppression de tâches. Cependant, dans le cas d'ajout de tâches, des contraintes temps-réel peuvent être violées et le système peut ne pas être faisable, i.e, non ordonnançable. Dans le cas d'un système rechargeable périodiquement, la contrainte énergétique est une contrainte supplémentaire qui doit être respectée. Le système est dit faisable en énergie si et seulement s'il peut poursuivre son exécution jusqu'à la prochine recharge sans épuisement de la batterie. C'est dans ce contexte que notre travail s'inscrit.

5.2 Contributions

L'objectif principal de la thèse a donc été de proposer une stratégie de placement et d'ordonnancement de tâches communicantes permettant d'exécuter des applications temps-réel sur une architecture contenant des coeurs hétérogènes, en vue de gérer la quantité d'énergie disponible dans la batterie et de garantir le fonctionnement du système jusqu'à la prochaine recharge. Dans cette thèse, nous avons choisi d'aborder cette problématique de façon incrémentale pour traiter progressivement les problèmes liés aux contraintes temps-réel, énergétique et communication. Tout d'abord, nous nous intéressons plus particulièrement à l'ordonnancement des tâches pour l'architecture mono-coeur. Puis nous l'avons étendu pour traiter le cas de l'ordonnancement pour l'architecture multi-coeurs homogènes. Finalement, une extension de ce dernier a été réalisée afin d'arriver à l'objectif principal qui est le placement et l'ordonnancement des tâches pour les architectures hétérogènes. Notre manuscrit de thèse est composé de trois principaux chapitres :

Dans le **chapitre 2**, un état de l'art a été présenté sur les systèmes embarqués temps-réel ainsi que les différents travaux traitant la problématique d'ordonnancement sur les architectures mono et multi-coeurs. Nous avons mentionné, dans ce chapitre, que l'ensemble de nos travaux sont basés sur le modèle de tâches proposé par par Marinoni Mauro et Buttazzo Giorgio [14] qui est caractérisé par l'élasticité des tâches et qui est très utilisé surtout dans les systèmes temps-réel à contraintes souples [2, 7, 9, 15–20].

Les principales contributions du **chapitre 3** sont :

- Proposition d'une stratégie d'ordonnancement de tâches pour les architectures mono-cœur reconfigurables dynamiquement. Elle permet de garantir la faisabilité du système après la violation de l'une des contraintes temps-réel et/ou énergétiques. L'objectif de la stratégie proposée est d'assurer l'ordonnancement des tâches, sous les contraintes précitées, en adaptant leurs temps d'exécution et leurs périodes.
- Réalisation d'un outil de simulation qui permet d'évaluer la stratégie proposée par rapport à l'algorithme proposé par Wang et al. [2, 15]. Les résultats montrent que la stratégie proposée permet de réduire le coût de modification des paramètres par rapport à l'approche dans [2, 15]. En terme de gain et sur un ensemble de systèmes générés aléatoirement, notre stratégie offre 27% de gain de coût de modification par rapport à l'approche proposée par Wang et al. [2, 15].

Concernant le **chapitre 4**, nous avons adressé la problématique du placement et d'ordonnancement de tâches sur une architecture multi-cœurs et une nouvelle contrainte a été traité qui est la contrainte de communication. Les principales contributions de ce chapitre sont :

- Proposition d'une stratégie d'ordonnancement adressant les architectures multi-cœurs homogènes. Dans cette architecture, les tâches sont ordonnancées sous les trois contraintes suivantes : temps-réel, énergétique et communication,
- Proposition d'une extension de la stratégie précédente qui s'adresse aux architectures multi-cœurs hétérogènes. Elle propose de placer et d'ordonnancer les tâches après un événement de configuration ou de reconfiguration tout en garantissant la faisabilité du système. Les résultats expérimentaux sur notre stratégie montrent qu'elle offre un gain du coût de communication de 11% par rapport à la stratégie proposée par Govil et al. [21] et de 7% par rapport à l'algorithme proposé par Bhardwaj et al. [22]. Par ailleurs, nous avons utilisé une autre métrique de comparaison qui est le taux de rejet des tâches et nous avons montré que la stratégie proposée permet de réduire le taux de rejet de 62% par rapport à l'algorithme [21] et de 59% par rapport à celui présenté dans [22].

5.3 Perspectives

À l'issue de cette thèse, nous envisageons les perspectives suivantes :

- **Amélioration de la qualité de service du système :** Le cas d'un scénario de reconfiguration qui supprime des tâches n'est pas traité dans cette thèse puisqu'il ne viole pas le(s) contraintes du système. Bien que ce cas ne risque pas de rendre le système non faisable, il serait intéressant d'envisager la réévaluation des paramètres des tâches après le scénario de suppression. Autrement dit, en profitant du cas de suppression de tâches pour améliorer la performance du système. Cela peut se faire par la minimisation de la dégradation de qualité de services du système tout en réévaluant les paramètres de tâches après chaque reconfiguration.

  En prenant l'étude de cas *RE-CONF* qui est composée initialement de 4 tâches (Section 3.3.1), après le premier scénario de reconfiguration qui a ajouté τ_5 et

τ_6, le système devient non ordonnançable puisque son taux d'utilisation est égal à 1.19 > 1. Après l'application de l'heuristique A en allongeant les périodes de tâches, le taux d'utilisation a été baissé à une valeur inférieure à 1 et le système redevient ordonnançable.

Maintenant, nous supposons que le système subit une deuxième reconfiguration qui supprime deux tâches (disons τ_5 et τ_6), le système revient donc à son état initial qui est faisable, mais les caractéristiques de tâches restent modifiées. En effet, dans certains cas particuliers, on peut avoir un système faisable dégradé alors qu'il peut être faisable mais non dégradé si on réévalue les caractéristiques de ses tâches. Conséquemment, un axe d'amélioration de la qualité de service consisterait à réévaluer les paramètres de tâches après chaque scénario de reconfiguration.

- **Implémentation dans un OS :** Les travaux de recherche futurs reposent sur l'intégration des stratégies développées dans les chapitres 3 et 4 dans un OS comme TinyOS [6, 108] qui est un système d'exploitation gratuit et open-source dédié aux capteurs. Les applications pour TinyOS sont écrites en nesC [109] (Network Embedded System C), une extension de C. Nos stratégies peuvent être également intégrées au sein du *middleware* reconfigurable développé par Khemaissia Imène [29]. Il est basé sur la plate-forme OSEK/VDX et est capable de supporter la reconfiguration des tâches élastiques. Dans tous les cas, pour être applicable, notre méthodologie devrait avoir accès à l'information du niveau d'énergie dans la batterie et cela peut se faire en développant un module logiciel s'interfaçant avec le matériel pour lui fournir cette information.

- **Prise en compte des tâches apériodiques :** Dans ce travail, nous n'avons traité que le cas des tâches périodiques. Bien que la majorité des applications temps-réel soient composées de tâches périodiques, il serait intéressant d'étudier le comportement de nos approches en prenant en compte les tâches apériodiques afin de répondre à tous les besoins des concepteurs d'applications temps-réel. Pour les systèmes de sécurité, certaines tâches apériodiques peuvent être critiques (comme les alarmes) et le système se doit de fournir les solutions qui garantissent également les contraintes pour ce genre de tâches.

- **Réalisation pratique de la partie frame-packing au niveau du support de communication :** Après avoir testé théoriquement la faisabilité du support de communication, il serait intéressant d'étendre ce travail pour réaliser la fragmentation des messages et la construction des trames afin de valider l'approche en pratique.

5.4 Publications

Revues Internationales :

- **A. Gammoudi**, A. Benzina, M. Khalgui and D. Chillet. Energy-Efficient Scheduling of Real-Time Tasks in Reconfigurable Homogeneous Multi-Core Platforms. *IEEE Transactions on Systems, Man, and Cybernetics : Systems* (en révision mineure).

Conférences Internationales avec comité de lecture :

- **Aymen Gammoudi**, Daniel Chillet, Mohamed Khalgui and Adel Benzina. Mapping of Periodic Tasks in Reconfigurable Heterogeneous Multi-Core Platforms. *13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, Funchal, Madeira, Portugal, March 23-24, 2018.
- **Aymen Gammoudi**, Adel Benzina, Mohamed Khalgui and Daniel Chillet. Real-Time Scheduling of Reconfigurable Battery-Powered Multi-Core Platforms. *28th International Conference on Tools with Artificial Intelligence (ICTAI)*, San Jose, USA, November 06-08, 2016.
- **Aymen Gammoudi**, Adel Benzina, Mohamed Khalgui, Daniel Chillet and Aicha Goubaa. Reconf-Pack : A Simulator for Reconfigurable Battery-Powered Real-Time Systems. *30th European and Modiling Conference (ESM)*, University of Las Palmas, Spain, October 26-28, 2016.
- **Aymen Gammoudi**, Adel Benzina, Mohamed Khalgui and Daniel Chillet. New Reconfigurable Middleware for Adaptive RTOS in Ubiquitous Devices. *10th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (Ubicomm)*, Venice, Italie, Octobre 09-13, 2016.

- **Aymen Gammoudi**, Adel Benzina, Mohamed Khalgui and Daniel Chillet. New Pack Oriented Solutions for Energy-Aware Feasible Adaptive Real-Time Systems. *14th International Conference on Intelligent Software Methodologies Tools, and Techniques (SoMeT)*, Naples, Italie, September 15-17, 2015.

Conférences Nationales avec Comité de Lecture :

- **Aymen Gammoudi**, Adel Benzina, Mohamed Khalgui and Daniel Chillet. Feasible Real-Time Scheduling under Memory and Energy Constraints. *Les premières journées doctorales de l'Ecole Polytechnique de Tunisie (JDEPT)*, La Marsa, Tunis, October 22-23, 2015.

Posters :

- **Aymen Gammoudi**, Adel Benzina, Mohamed Khalgui and Daniel Chillet. New Pack Oriented Solutions for Energy-Aware Feasible Adaptive Real-Time Systems. *1ère Doctoriale Valorisation de la Recherche de l'INSAT*, INSAT, Tunisie, May 14, 2015.

Bibliographie

[1] Y Atat. *Conception de haut niveau des MPSoCs à partir d'une spécification Simulink : Passerelle entre la conception d'algorithmes et la conception d'architectures.* PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2007.

[2] Xi Wang, Imen Khemaissia, Mohamed Khalgui, ZhiWu Li, Olfa Mosbahi, and MengChu Zhou. Dynamic low-power reconfiguration of real-time systems with periodic and probabilistic tasks. *IEEE Transactions on Automation Science and Engineering*, 12(1) :258–271, 2015.

[3] Hui Zhang. *Gestion de l'énergie renouvelable et ordonnancement temps réel dans les systèmes embarqués.* PhD thesis, Université de Nantes, 2012.

[4] Younès Chandarli. *Gestion de l'énergie renouvelable et ordonnancement temps réel dans les systèmes embarqués.* PhD thesis, Université Paris-Est, 2014.

[5] Maryline Chetto and Hussein El Ghor. Real-time scheduling of periodic tasks in a monoprocessor system with a rechargeable battery. In *The 30th IEEE Real-Time Systems Symposium*, page 45, 2009.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *ACM SIGOPS operating systems review*, 34(5) :93–104, 2000.

[7] Xi Wang, ZhiWu Li, and WM Wonham. Dynamic multiple-period reconfiguration of real-time scheduling based on timed des supervisory control. *IEEE Transactions on Industrial Informatics*, 12(1) :101–111, 2016.

[8] Laurent George and Pierre Courbin. Reconfiguration of uniprocessor sporadic real-time systems : the sensitivity approach. *book chapter in IGI-Global Knowledge on*

Reconfigurable Embedded Control Systems : Applications for Flexibility and Agility, pages 167–189, 2010.

[9] Wiem Housseyni, Olfa Mosbahi, Mohamed Khalgui, and Maryline Chetto. Real-time scheduling of reconfigurable distributed embedded systems with energy harvesting prediction. In *Distributed Simulation and Real Time Applications (DS-RT), 2016 IEEE/ACM 20th International Symposium on*, pages 145–152. IEEE, 2016.

[10] Jiafeng Zhang, Mohamed Khalgui, Zhiwu Li, Olfa Mosbahi, and Abdulrahman M Al-Ahmari. R-tnces : A novel formalism for reconfigurable discrete event control systems. *IEEE Transactions on Systems, Man, and Cybernetics : Systems*, 43(4) : 757–772, 2013.

[11] Martijn N Rooker, Christoph Sünder, Thomas Strasser, Alois Zoitl, Oliver Hummer, and Gerhard Ebenhofer. Zero downtime reconfiguration of distributed automation systems : The εcedac approach. In *Holonic and Multi-Agent Systems for Manufacturing*, pages 326–337. Springer, 2007.

[12] Ling Li, Shancang Li, and Shanshan Zhao. Qos-aware scheduling of services-oriented internet of things. *IEEE Transactions on Industrial Informatics*, 10(2) : 1497–1505, 2014.

[13] Pavel Vrba and Vladimír Marik. Capabilities of dynamic reconfiguration of multiagent-based industrial control systems. *IEEE Transactions on Systems, Man, and Cybernetics-Part A : Systems and Humans*, 40(2) :213–223, 2010.

[14] Mauro Marinoni and Giorgio Buttazzo. Elastic dvs management in processors with discrete voltage/frequency modes. *IEEE Transactions on industrial informatics*, 3 (1) :51–62, 2007.

[15] Xi Wang, Mohamed Khalgui, and Zhiwu Li. Dynamic low power reconfigurations of real-time embedded systems. pages 415–420, 2011.

[16] Hamza Gharsellaoui, Mohamed Khalgui, and Samir Ben Ahmed. New optimal preemptively scheduling for real-time reconfigurable sporadic tasks based on earliest deadline first algorithm. *International Journal of Advanced Pervasive and Ubiquitous Computing (IJAPUC)*, 4(2) :65–81, 2012.

[17] Imen Khemaissia, Olfa Mosbahi, and Mohamed Khalgui. Reconfigurable can in real-time embedded platforms. In *Informatics in Control, Automation and Robotics (ICINCO), 2014 11th International Conference on*, volume 1, pages 355–362. IEEE, 2014.

[18] Mohamed Khalgui, Olfa Mosbahi, and Zhiwu Li. Runtime reconfigurations of embedded controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1) :14, 2013.

[19] Imen Khemaissia, Olfa Mosbahi, Mohamed Khalgui, and Zhiwu Li. Crmpsoc : New solution for feasible reconfigurable mpsoc. In *International Conference on Software Technologies*, pages 175–198. Springer, 2016.

[20] Wiem Housseyni, Olfa Mosbahi, Mohamed Khalgui, and Maryline Chetto. Real-time scheduling of sporadic tasks in energy harvesting distributed reconfigurable embedded systems. In *Computer Systems and Applications (AICCSA), 2016 IEEE/ACS 13th International Conference of*, pages 1–8. IEEE, 2016.

[21] Kapil Govil. A smart algorithm for dynamic task allocation for distributed processing environment. *International Journal of Computer Applications*, 28(2) :13–19, 2011.

[22] Poornima Bhardwaj and Vinod Kumar. An effective load balancing task allocation algorithm using task clustering. *International Journal of Computer Applications*, 77(7), 2013.

[23] Hermann Kopetz. *Real-time systems : design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[24] Jean Paul Calvez, Alan Wyche, and Charles Edmundson. *Embedded real-time systems*. J. Wiley, 1993.

[25] Federico Angiolini, Jianjiang Ceng, Rainer Leupers, Federico Ferrari, Cesare Ferri, and Luca Benini. An integrated open framework for heterogeneous mpsoc design space exploration. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1145–1150. European Design and Automation Association, 2006.

[26] Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Heuristics for dynamic task mapping in noc-based heterogeneous mpsocs. In *Rapid System Prototyping,*

2007. RSP 2007. 18th IEEE/IFIP International Workshop on, pages 34–40. IEEE, 2007.

[27] Philippe Grosse. *Gestion dynamique des tâches dans une architecture micro-éléctronique intégrée, à des fins de basse consommation.* PhD thesis, École normale supérieure (Lyon), 2007.

[28] Xi Wang, Mohamed Khalgui, and Zhiwu Li. Dynamic low power reconfigurations of embedded real-time systems. In *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems, Portugal*, volume 6. Citeseer, 2010.

[29] Imène Khemaissia. *Reconfiguration des systèmes embarqués sous contraintes temps réel et d'énergie.* PhD thesis, Université de TUNIS EL-MANAR, 2016.

[30] Imen Khemaissia, Olfa Mosbahi, Mohamed Khalgui, and Zhiwi Li. New methodology for feasible reconfigurable real-time network-on-chip noc. In *Proc. 11th Int. Conf. Softw. Eng. App on*, 2016.

[31] Hamza Chniter, Mohamed Khalgui, and Fethi Jarray. Adaptive embedded systems : New composed technical solutions for feasible low-power and real-time flexible os tasks. In *Informatics in Control, Automation and Robotics (ICINCO), 2014 11th International Conference on*, volume 1, pages 92–101. IEEE, 2014.

[32] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks : a survey. *Computer networks*, 38(4) :393–422, 2002.

[33] Yasmina Abdeddaïm, Younès Chandarli, Robert I Davis, and Damien Masson. Schedulability analysis for fixed priority real-time systems with energy-harvesting. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 311. ACM, 2014.

[34] Maryline Chetto and Audrey Queudet. A note on edf schedulingfor real-time energy harvesting systems. *IEEE Transactions on Computers*, 63(4) :1037–1040, 2014.

[35] Hussein EL Ghor, Maryline Chetto, and Rafic Hage Chehade. A real-time scheduling framework for embedded systems with environmental energy harvesting. *Computers & Electrical Engineering*, 37(4) :498–510, 2011.

[36] Hussein El Ghor, Maryline Chetto, and Rafic Hage Chehade. A nonclairvoyant real-time scheduler for ambient energy harvesting sensors. *International Journal of Distributed Sensor Networks*, 9(5) :732652, 2013.

[37] Hamza Chniter, Fethi Jarray, and Mohamed Khalgui. Combinatorial approaches for low-power and real-time adaptive reconfigurable embedded systems. In *PECCS*, pages 151–157, 2014.

[38] Fethi Jarray, Hamza Chniter, and Mohamed Khalgui. New adaptive middleware for real-time embedded operating systems. In *Computer and Information Science (ICIS), 2015 IEEE/ACIS 14th International Conference on*, pages 610–618. IEEE, 2015.

[39] Hamza Gharsellaoui, Atef Gharbi, Mohamed Khalgui, and Samir Ben Ahmed. Feasible automatic reconfigurations of real-time os tasks. In *Handbook of Research on Industrial Informatics and Manufacturing Intelligence : Innovations and Solutions*, pages 390–414. IGI Global, 2012.

[40] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[41] John A Stankovic and Krithi Ramamritham. *Hard real-time systems : tutorial*. [sn], 1988.

[42] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on software engineering*, 15(10) :1261, 1989.

[43] Giorgio C Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3) :289–302, 2002.

[44] Mauro Marinoni and Giorgio Buttazzo. Adaptive dvs management through elastic scheduling. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, volume 2, pages 7–pp. IEEE, 2005.

[45] Giorgio C Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic task model for adaptive rate control. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 286–295. IEEE, 1998.

[46] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. Riot os : Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[47] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James H Anderson, and Sanjoy K Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms., 2004.

[48] Liliana Cucu. *Ordonnancement non préemptif et condition d'ordonnançabilité pour systèmes embarqués à contraintes temps réel.* PhD thesis, Université Paris Sud-Paris XI, 2004.

[49] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2) :244–257, 1989.

[50] Wesley W. Chu and M-T Lan. Task allocation and precedence relations for distributed real-time systems. *IEEE transactions on Computers*, 36(6) :667–679, 1987.

[51] Jia Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on software engineering*, 19 (2) :139–154, 1993.

[52] Pascal Richard, Francis Cottet, and Michaël Richard. On-line scheduling of real-time distributed computers with complex communication constraints. In *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on*, pages 26–34. IEEE, 2001.

[53] Mien Forget, Emmanuel Grolleau, Claire Pagetti, and Pascal Richard. Dynamic priority scheduling of periodic tasks with extended precedences. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8. IEEE, 2011.

[54] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3) :307–338, 2011.

[55] Joseph Y-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4) :237–250, 1982.

[56] James R Jackson. Scheduling a production line to minimize maximum tardiness. Technical report, CALIFORNIA UNIV LOS ANGELES NUMERICAL ANALYSIS RESEARCH, 1955.

[57] Aloysius Ka-Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment.* PhD thesis, Massachusetts Institute of Technology, 1983.

[58] Cottet Francis, Delacroix Joelle, Kaiser Claude, and Mammeri Zoubir. Ordonnancement temps réel-cours et exercices corrigés. Technical report, 2000.

[59] Franck Bimbard. *Dimensionnement temporel de systèmes embarqués : application à OSEK.* PhD thesis, Paris, CNAM, 2007.

[60] Laurent George, Nicolas Rivierre, and Marco Spuri. *Preemptive and non-preemptive real-time uniprocessor scheduling.* PhD thesis, Inria, 1996.

[61] Sungwook Kim. Adaptive online voltage scaling scheme based on the nash bargaining solution. *ETRI Journal*, 33(3) :407–414, 2011.

[62] Tetsuo Yokoyama, Gang Zeng, Hiroyuki Tomiyama, and Hiroaki Takada. Static task scheduling algorithms based on greedy heuristics for battery-powered dvs systems. *IEICE transactions on information and systems*, 93(10) :2737–2746, 2010.

[63] Junlong Zhou, Jianming Yan, Kun Cao, Yanchao Tan, Tongquan Wei, Mingsong Chen, Gongxuan Zhang, Xiaodao Chen, and Shiyan Hu. Thermal-aware correlated two-level scheduling of real-time tasks with reduced processor energy on heterogeneous mpsocs. *Journal of Systems Architecture*, 2017.

[64] Mohamed Khalgui, Olfa Mosbahi, Zhiwu Li, and Hans-Michael Hanisch. Reconfigurable multiagent embedded control systems : From modeling to implementation. *IEEE Transactions on Computers*, 60(4) :538–551, 2011.

[65] Nadine Abdallah. *Partitionnement temps réel multiprocesseur sous contraintes de qualité de service et d'énergie.* PhD thesis, Université de Nantes, 2014.

[66] WA Horn. Some simple scheduling algorithms. *Naval Research Logistics (NRL)*, 21(1) :177–185, 1974.

[67] Audrey Queudet, Nadine Abdallah, and Maryline Chetto. Kts : a real-time mapping algorithm for noc-based many-cores. *The Journal of Supercomputing*, pages 1–17, 2017.

[68] Naman Govil, Rahul Shrestha, and Shubhajit Roy Chowdhury. Pgma : An algorithmic approach for multi-objective hardware software partitioning. *Microprocessors and Microsystems*, 54 :83–96, 2017.

[69] Jia Huang, Andreas Raabe, Christian Buckl, and Alois Knoll. A workflow for runtime adaptive task allocation on heterogeneous mpsocs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[70] Jia Huang, Christian Buckl, Andreas Raabe, and Alois Knoll. Energy-aware task allocation for network-on-chip based heterogeneous multiprocessor systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 447–454. IEEE, 2011.

[71] Suleyman Tosun. New heuristic algorithms for energy aware application mapping and routing on mesh-based nocs. *Journal of Systems Architecture*, 57(1) :69–78, 2011.

[72] Jingcao Hu and Radu Marculescu. Energy-and performance-aware mapping for regular noc architectures. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 24(4) :551–562, 2005.

[73] Jakob Puchinger and Günther R Raidl. Combining metaheuristics and exact algorithms in combinatorial optimization : A survey and classification. In *International Work-Conference on the Interplay Between Natural and Artificial Computation*, pages 41–53. Springer, 2005.

[74] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica : Journal of the Econometric Society*, pages 497–520, 1960.

[75] Gen-Huey Chen and Jyr-Shiarn Yur. A branch-and-bound-with-underestimates algorithm for the task assignment problem with precedence constraint. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 494–501. IEEE, 1990.

[76] El-Ghazali Talbi. *Metaheuristics : from design to implementation*, volume 74. John Wiley & Sons, 2009.

[77] Dar-Tzen Peng, Kang G. Shin, and Tarek F. Abdelzaher. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12) :745–758, 1997.

[78] John R Birge. Linear programming : Foundations and extensions. *IEEE Transactions*, 31(3) :278–278, 1999.

[79] George B Dantzig and Mukund N Thapa. *Linear programming 2 : theory and extensions*. Springer Science & Business Media, 2006.

[80] Berkelaar M, Eikland K, Notebaert P, and al. Open source (mixed-integer) linear programming system. In *Eindhoven U. of Technology*, 2004.

[81] CPLEX. Ibm ilog cplex optimize. In *http ://www-03.ibm.com/software/products/en/ibmilogcpleoptistud/*, 2013.

[82] Falou Ndoye. *Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation*. PhD thesis, Université Paris Sud-Paris XI, 2014.

[83] Navonil Chatterjee, Suraj Paul, Priyajit Mukherjee, and Santanu Chattopadhyay. Deadline and energy aware dynamic task mapping and scheduling for network-on-chip based multi-core platform. *Journal of Systems Architecture*, 74 :61–77, 2017.

[84] Suleyman Tosun, Ozcan Ozturk, and Meltem Ozen. An ilp formulation for application mapping onto network-on-chips. In *Application of Information and Communication Technologies, 2009. AICT 2009. International Conference on*, pages 1–5. IEEE, 2009.

[85] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9) :1175–1185, 1990.

[86] Thomas L Adam, K. Mani Chandy, and JR Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12) :685–690, 1974.

[87] Carolyn L McCreary, AA Khan, JJ Thompson, and ME McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 446–451. IEEE, 1994.

[88] Michael A. Palis, Jing-Chiou Liou, and David S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1) :46–55, 1996.

[89] Wesley W Chu, Leslie J Holloway, Min-Tsung Lan, and Kemal Efe. Task allocation in distributed data processing. *IEEE computer*, 13(11) :57–69, 1980.

[90] Apostolos Gerasoulis, Sesh Venugopal, and Tao Yang. Clustering task graphs for message passing architectures. *ACM SIGARCH Computer Architecture News*, 18 (3b) :447–456, 1990.

[91] Kato Shinpei and Nobuyuki Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 441–450. IEEE, 2007.

[92] Gang Quan and Xiaobo Sharon Hu. Minimal energy fixed-priority scheduling for variable voltage processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(8) :1062–1071, 2003.

[93] Aymen Gammoudi, Adel Benzina, Mohamed Khalgui, and Daniel Chillet. New pack oriented solutions for energy-aware feasible adaptive real-time systems. In *International Conference on Intelligent Software Methodologies, Tools, and Techniques (SoMeT)*, pages 73–86. Springer, 2015.

[94] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The turbo diaries : Application-controlled frequency scaling explained. In *USENIX Annual Technical Conference*, pages 193–204, 2014.

[95] Youngsoo Shin and Kiyoung Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 134–139. ACM, 1999.

[96] Aymen Gammoudi, Adel Benzina, Mohamed Khalgui, Daniel Chillet, and Aicha Goubaa. Reconf-pack : A simulator for reconfigurable battery-powered real-time systems. In *30th European Simulation and Modelling Conference (ESM)*, 2016.

[97] Aymen Gammoudi, Adel Benzina, Daniel Chillet, and Mohamed Khalgui. New reconfigurable middleware for adaptive rtos in ubiquitous devices. In *10th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (Ubicomm)*, 2016.

[98] FSM Lab. Getting-started-with-rtlinux. 42, 2001.

[99] Imène Benkermi. *Modèle et algorithme d'ordonnancement pour architectures reconfigurables dynamiquement.* PhD thesis, Rennes 1, 2007.

[100] Thierry Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, PARIS 11, ORSAY, 2000.

[101] Houssine Chetto, Maryline Silly, and T Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3) :181–194, 1990.

[102] Bach Duy Bui, Rodolfo Pellizzoni, and Marco Caccamo. Real-time scheduling of concurrent transactions in multidomain ring buses. *IEEE Transactions on Computers*, 61(9) :1311–1324, 2012.

[103] Kristian Sandstrom, C Norstom, and Magnus Ahlmark. Frame packing in real-time communication. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 399–403. IEEE, 2000.

[104] Ricardo Santos Marques, Nicolas Navet, and Françoise Simonot-Lion. Frame packing under real-time constraints. *IFAC Proceedings Volumes*, 36(13) :181–188, 2003.

[105] Aymen Gammoudi, Adel Benzina, Mohamed Khalgui, and Daniel Chillet. Real-time scheduling of reconfigurable battery-powered multi-core platforms. In *Tools*

with Artificial Intelligence (ICTAI), 2016 IEEE 28th International Conference on, pages 121–129. IEEE, 2016.

[106] Hyung Gyu Lee, Naehyuck Chang, Umit Y Ogras, and Radu Marculescu. On-chip communication architecture exploration : A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3) :23, 2007.

[107] Theodoros K Konstantakopoulos. *Energy scalability of on-chip interconnection networks*. PhD thesis, Massachusetts Institute of Technology, 2007.

[108] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos : An operating system for sensor networks. *Ambient intelligence*, 35 :115–148, 2005.

[109] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language : A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.